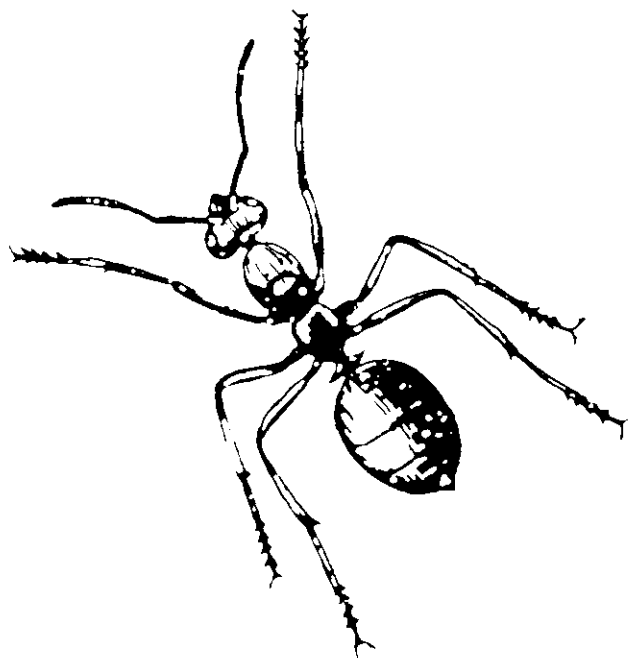# TMON

ICOM Simulations Inc.

formerly TMQ Software, Inc.

- Interactive, Multi-window Monitor/Debugger for the Macintosh™

- Works on Both 128K and 512K Versions of the Machine

- An absolute must for any serious development on the Macintosh.™

Programmed by Waldemar Horwat exclusively for TMQ Software, Inc.

TMON

TMQ Software

Waldemar Horwat

Monitor version 2.585

April 19, 1985

This is a technical manual. It is not intended in any way to be a tutorial.

Table of Contents

The bold items indicate chapters and sections which should be read in order to begin using the Monitor. The other sections provide information which is helpful to fully exploit the Monitor's features. The **Creating Your Own User Functions** chapter is for advanced programmers only. Read the **Macintosh Memory Management** chapter if you are unfamiliar with the Macintosh operating system routines.

Introduction

TMON is an assembly language monitor/debugger for the Macintosh personal computer. It will work on both 128K and 512K versions of the machine. Among the features included are:

- A *fast* implementation of windows on the screen that has these advantages:
    - Information is not lost when it scrolls off the screen.
    - Registers, breakpoints, program code, subroutines, data, stack, heaps, and resource files can all be examined at the same time.
    - Sections of code can be cross-referenced.
    - Windows can be scrolled up or down.
    - Disassembly and dump windows can be anchored to registers.
    - Instruction and effective addresses in disassembly windows are identified using labels.
- Windows updated *continuously*. When one window is changed, the other windows instantly reflect the change.
- An interactive 68000 assembler/disassembler.
    - Includes reverse scrolling of disassembly windows.
    - A000 traps are displayed by their names, not numbers.
    - Labels may be used both by the disassembler and in assembling.
- Label and symbol capabilities.
    - Labels may be used in any expression.
    - Labels may be recognized automatically from routine names in code.
    - Labels may be loaded from .Map files.
    - Labels may be entered directly from TMON as either absolute or resource-relative.
    - Names of the A000 traps are used as labels during examination of ROM routines.
- An interactive hexadecimal and ASCII memory dump and change.
- File windows which identify all resources in all open resource files.
    - Resources not currently in memory are also displayed.
    - Resource flags are shown in an easy to read format.
    - Resource types, IDs, names, references, and handles are shown when appropriate.
    - Information displayed is checked for consistency.
- Heap windows displaying the contents of the application or system heaps.
    - The location, size, and type of all heap objects are displayed.
    - The addresses of handles and flags are displayed for relocatable objects.
    - The resource ID, type, and file are displayed for objects which are resources.
    - Windows, controls, window regions, scraps, and various other heap objects are identified. A user routine can be made to identify other heap objects when appropriate.
    - Information displayed is checked for consistency.
- Register windows which display and allow changing of all 68000 registers.
    - The flags are displayed in an easy to read format.
- TMON allows saving, loading, and exchanging registers with an alternate register set.
- Converting numbers and expressions between hexadecimal, decimal, binary, ASCII, A000 trap names, and labels.
- Use of expressions involving hexadecimal, decimal, and binary numbers, ASCII values, addition, subtraction, multiplication, division, boolean operations, indirection, parenthesis, A000 trap names, and labels.
- Up to seven breakpoints.
- Single-step execution of programs.
    - All stepping and tracing features work in ROM.
    - Tracing into A000 traps is possible.
    - A convenient function for skipping subroutines during tracing is included.

- Searching a block of memory for a 1, 2, 3, or 4 byte value.
  - Word-aligned, as well as byte-aligned searches.
- Block move, compare, fill, and checksum.
- Interception of almost all exceptions and system errors.
- Interception of any A000 traps upon request.
- Interception of program at a specific point when interrupt is pressed.
- Quiet recording of all or specific A000 traps which allows the course of execution of a program to be quickly traced. This function may also be used for performance analysis as it records the times of the A000 traps.
- A heap check function, which may be automatically run on A000 traps.
- A highly optimized heap scramble function, which may be automatically run on A000 traps as well. A purge option is available. The heap scramble clears the unused blocks, providing additional debugging security.
- A variable-length user area designed to allow customization of the monitor.
- Screen packing options for memory-tight applications.
- Printing to a printer, external computer, or terminal.
  - Any window on the screen can be printed.
  - Disassemblies and dumps of arbitrarily large blocks of memory can be printed.
  - Heap and resource file dumps can be printed.
  - XOn/XOff and DTR handshaking are supported.
  - Printing can be done from either port.
- Mouse unfreeze and vector refresh options, which may be very helpful after a program crashes.
- The ability to easily disable the more system-dependent Monitor functions like labels and heap identification in case these functions fail.
- A Monitor self-test run continuously to provide extra security.
- Choice of either system heap or high memory to load the Monitor allowing the Monitor to work with virtually all programs.
- The ability to quietly load the Monitor upon starting the Macintosh without any interaction.
- The capability of automatically patching the Monitor code via a user area routine.

the TMON disk will continue to be used, forcing you to keep the TMON disk in one of the drives on a two-drive Macintosh or swap disks on a single-drive Macintosh.

The Launch function assumes that if there is a System file on the target disk, a Finder is also present. If the disk contains a system file but no Finder, the Macintosh will crash after you leave the program you launched. The Macintosh will also crash later if there is not enough memory to run the Finder.

Quit

Quit exits to the Finder.

On 128K Macintoshes any attempt to run the Finder if the Monitor Size is more than about 26000 bytes will fail. See the **Crash While Exiting to Finder** section in the **Possible Problem Areas** chapter. If you do have a 128K Macintosh, it is both much faster and less risky to use the Launch function to start the program you want to debug.

Monitor. In this case you will see a message for a few seconds; afterwards, the Finder screen will appear. More information about this is available in the Auto-Quit section of the Configuration chapter. For now, just hold down Shift, Option, ⌘, or the mouse button while TMON is loading.

The Main Menu

The *second* Main Menu will appear like this:

```
┌─────────────────────────────────────────────────────┐
│ ┌─────────────────────────────────────────────────┐ │
│ │                                                   │ │
│ │  TMON version 2.585         ╭─Configure─╮        │ │
│ │                             ╰───────────╯        │ │
│ │  Written by                 ╭──Monitor──╮        │ │
│ │  Waldemar Horwat            ╰───────────╯        │ │
│ │                             ╭─Monitor…──╮        │ │
│ │  ©1985 TMQ Software, Inc.   ╰───────────╯        │ │
│ │                             ╭──Launch───╮        │ │
│ │  1110 Lake-Cook Rd.         ╰───────────╯        │ │
│ │  Buffalo Grove, IL 60090    ╭───Quit────╮        │ │
│ │                             ╰───────────╯        │ │
│ └─────────────────────────────────────────────────┘ │
└─────────────────────────────────────────────────────┘
```
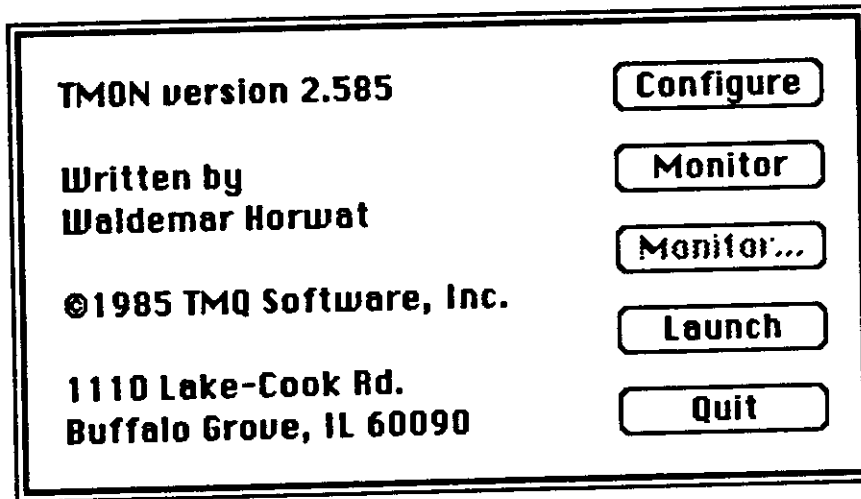
At this point you have a choice of five options. You may execute them by either clicking on the buttons or by typing on the keyboard: C for Configure, M for Monitor, . (period) for Monitor..., L for Launch, and Q for Quit. Monitor... is disabled in the *second* Main Menu, and Configure is disabled in the *first* Main Menu.

Monitor
Monitor...

When used from the *first* Main Menu, Monitor will load and enter the Monitor using either the file called User Area on the disk, or, if there is none, the hidden user area in TMON to boot the Monitor. More information on user areas can be found throughout this manual. This option is described in more detail in the next chapter.

Monitor... is the same as Monitor except that it allows a user area other than the default one to be selected and used to boot the Monitor. Clicking Open will start the Monitor, while clicking Cancel will return to the Main Menu. Hereafter these two options will be described together.

In the *second* Main Menu the Monitor button will re-initialize the Monitor and reenter it without reloading it from the disk. Use the interrupt button to reenter the Monitor without re-initializing it.

➪   Do not use this button to re-initialize the Monitor if breakpoints have been set because the Monitor will forget about them even though they remain set. You will then be unable to clear the breakpoints.

Launch

Launch is a quick way to leave TMON and execute another application. After you click this button you will get a listing of all applications on your disks. Choose the application you want and click Launch to execute it. Click Cancel if you change your mind and don't want to execute an application.

If there is a system file on the disk with the application to be launched, TMON will switch to that disk's system file. This means that once you remove the TMON disk from the disk drive you will not need it again until you turn off or reset the Macintosh. On the other hand, if there is no system file on the application disk, the system file on

Entering the Monitor

The Monitor

When the Monitor starts, the entire screen clears to gray except for a list of commands in a small bar at the top of the screen and a welcome message near the top of the screen. Now you can use any of the Monitor's features which are explained below and in the succeeding chapters.

The Menu Bar

There are fourteen commands in the menu bar. To select a command, position the mouse over one of the commands and click the mouse button. It is also possible to use the keyboard equivalent of any of the commands in the menu bar. For that hold down the ⌘ key and type the first letter of the command.

☞   Note that the menu at the top of the screen behaves differently than the standard Macintosh menus; it is more similar to a collection of buttons. If you change your mind and don't want to select any function in the menu bar, move the mouse below the menu bar and then release it.

☞   One command does not appear in the menu bar. It is the Mouse Unfreeze command. The only way to invoke it is to type ⌘M.

Windows

The commands Dump, Assembly, Breakpoints, Registers, File, Heap, Number, Options, and User use windows to do their functions. Executing one of these commands, either by clicking in the menu bar or typing the keyboard equivalent, causes a corresponding window to appear or be uncovered. Breakpoints, Registers, Options, and User allow only one window each of their types to appear on the screen; Dump, Assembly, File, Heap, and Number allow as many as nine windows to appear simultaneously on the screen. To create additional windows of one of these types you can't just use the corresponding command, because that will just move the cursor to the top of the window. Instead, you have to hold down the Shift key while either clicking in the menu bar or typing the keyboard equivalent.

☞   Use the Shift key to create additional windows of the same type.

The windows behave similarly to normal Macintosh windows, although there are differences which will be explained here. All windows occupy the full width of the screen and therefore can't be moved horizontally. To close a window, click in the close box. To move it, press the mouse button down while the mouse is anywhere in the window (except in the close, resize, and scroll boxes), and drag the mouse up or down. Dump, Assembly, File, and Heap windows also have resize and scroll boxes on their right sides. Dragging the resize box in the lower right corner of the window will make the window bigger or smaller. Clicking one of the two scroll arrows will scroll the window one line, while holding down the mouse button there will repeatedly scroll the window. Note that the windows can be resized or scrolled even if they are partially covered by other windows, without being brought to the front. These windows can also be moved without being brought to the front by dragging them in the vertical area between the two scroll boxes. No more than nine windows can be open at any time.

☞   Windows can be dragged by pressing and dragging the mouse almost anywhere in the window, and not just in the title bar as with normal Macintosh windows.

Numbers

Whenever you are asked to enter a number, you may enter a number in either hexadecimal, decimal, binary, or ASCII, the value of a processor or special register, an indirection, an A000 trap name, a label, or any expression containing the above items. The default base is hexadecimal, but hexadecimal numbers may optionally be preceded by a dollar sign. Since spaces are often used to separate values, they may not be embedded in expressions except in ASCII values and labels. Decimal numbers are preceded with a period, and binary numbers are preceded with a percent sign. ASCII values must be enclosed by single quotes, and may contain single quotes themselves provided that the inside single quotes are doubled. (Example: To enter the string a'b, type 'a''b'.) Labels must be enclosed by double quotes, and must not contain double quotes themselves. Labels may be any length, but only the first eight characters are significant. See the Labels section for more information about labels.

A000 trap names are entered by typing an underscore (_) followed by the name of the trap. They may only be used if the A000 trap names have been loaded (see Configuration). The value generated by doing this will be the number of the trap as if the trap were entered into an Assembly window with one exception: the eleven traps that would assemble as $A1nn are converted to $A0nn. This is done to prevent confusion between traps like $A11E (_NewPtr) and $A91E (_TrackGoAway), especially because the user area routines take the trap numbers modulo $200. Another difference between the usage of the A000 traps in expressions and as Assembly window opcodes is that the Assembly window allows the value of bits 8 through 11 to be set by following the trap name with a number (or expression). Trap names in expressions do not allow that, but, since they are in expressions, the same effect may be achieved by adding the corresponding value to the trap name (Example: _Open+$200).

☞ Be aware that there are differences between the usage of the A000 trap names in expressions and as Assembly window opcodes. The trap names may also be used in expressions in Assembly windows; they are not considered opcodes. An example of an A000 trap name in an expression in an Assembly window is MOVE #_Open,D0.

The processor and special registers may be used in two ways. They can either be used to provide values in expressions or to be interpreted as actual registers. The second option will be called passing registers as variables. Consider this example to clarify this distinction: Suppose you type the MOVE A0,USP instruction into an assembly window. Also suppose that the current value of the user stack pointer (as displayed in the Registers window) is $12345678. If the assembler interprets the USP as a value, the instruction will actually be assembled as MOVE A0,$12345678. If, on the other hand, the USP is interpreted as a variable, the instruction will be recognized as MOVE A0,USP. In this particular example the assembler would choose the latter option. The exact rules on whether register values or variables are used are described below.

☞ Make sure that you understand the difference between value and variable references.

Some registers have different names depending on whether they are used as values or variables. If that is the case, there is no ambiguity. If the same name is used for both the value and the variable, the variable will be selected whenever possible. Only if that is not possible will the value be selected. For example, in the instruction MOVE A0,USP the USP is interpreted as a variable. On the other hand, in MOVE D0,USP the USP is a value since it cannot be a variable (There is no MOVE instruction from a data register to the user stack pointer). In most expressions references by variable are prohibited so the values will be used.

☞ The only places registers may be referenced by variable are in Assembly windows and anchoring.

☞   Resizing or scrolling a window does not automatically bring it to the front. Dragging a window also will not bring it to the front if the mouse is positioned in the area between the two scroll arrows.

The "active window" is a synonym for the frontmost window. It is a little more difficult to find which window is active in the Monitor than in the normal Macintosh windows. The active window, if there is one, is always the frontmost one and contains a flashing cursor. There is no active window if there is no window at all on the screen or if there is a special message displayed near the top of the screen. (An example of a special message is the welcoming message that appeared when you first entered the Monitor. Special messages can be distinguished by the fact that they have no close boxes and disappear upon the first keystroke or mouse button press.)

Refreshing of Windows

All windows on the screen are continuously being refreshed. This provides you with a unique real-time view of memory contents. The refreshing is faster if there are fewer and smaller windows on the screen. To see an example of refreshing, open a Dump window for locations $150-$180 or $800-$900. Also try opening an Assembly window starting at location $820.

☞   The line currently being edited (the line containing the cursor) is *never* refreshed. This is done to prevent the line from changing while it is edited.

Refreshing can sometimes become extremely slow due to the long time needed to display the information in some windows. This problem is especially noticeable when there is a large Assembly window showing a portion of ROM on the screen. The villain in this case is the label routine, as it takes more than 70% of the time used to refresh the screen. See the Labels section for more information about this problem.

The Cursor and the Editing Facilities

The cursor is the flashing bar on the screen. It appears inside the active window. You can move it to a different place by clicking the mouse over the new place. If you experiment with this a little, you will find that some windows will not let you put the cursor in some positions because these positions do not contain anything that could be changed. Everything that you type appears at the cursor's position. If you have a numeric keypad, you may also use the left and right arrows to move the cursor left and right on the same line.

To enter text (more on that later) just type it after positioning the cursor to the correct place. The Backspace key may be used to delete characters. Return and Enter are used to enter the data you have typed. Enter enters the entire line, including any data you may have typed to the right of the current cursor position. Return, on the other hand, only enters the line, starting from the leftmost position and up to the cursor. Clear on the numeric keypad may be used to erase the entire line without entering it. More information about whether to use Return or Enter appears in the individual command description sections which follow. You don't have to follow these rules if you don't want to; every place where one of these two keys may be used, the other may be used as well.

If you decide that you don't want to enter the text, move the mouse to some other line and click it there. This will undo all changes you may have made on the original line.

The Tab key may be used to move the cursor to the top left corner of the window. It is useful for entering addresses in Dump and Assembly windows, entering the Program Counter value in a Register window, etc.

The menu bar will flash after you have pressed Return or Enter if you made a mistake somewhere on the line.

☞   All text that is not between single quotes (') is converted to upper case; therefore, you may type anything except ASCII strings in either case without affecting the outcome. One exception to this rule is mentioned in the section on Dump windows.

`-1*-.10`	Evaluates as 10 ($A). Both minus signs here are unary. The period indicates that the second number is decimal.
`RA2+RD3*2`	The value of address register 2 plus twice the value of data register 3.
`22/2|2+F&'A'`	Same as `<22/2>|<2+<F&'A'>>`, which is $13.
`_Open`	Evaluates as $A000.
`_Read+$400`	Evaluates as $A402.
`_Alert`	Evaluates as $A985.
`_NewHandle`	Evaluates as $A022, although the Assembly window would assemble it as $A122. See the note earlier in the section.
`!_Open`	Gives the address of the ROM _Open routine.
`DSPT+16`	Gives the address of the 22nd byte of the ROM trap dispatcher.
`"WDef0000"+30`	Gives the address of the 48th byte of the WDEF 0 resource. See the next section for details on labels.

Labels

Labels make the Monitor a symbolic debugger. They allow you to work using names instead of often meaningless or arbitrary numbers. The Monitor's label facility is powerful, but, most of all, it is flexible. If you have some type of labels that are not recognized by the Monitor, and if you have the necessary expertise, you may write a user area function that will recognize those labels. If you are not an expert, someone else might do it.

There are two basic operations that can be done with labels: convert a label into an address ("evaluate a label") and convert an address into a label plus an offset ("recognize the address"). The first operation is done whenever you use a label in any expression; the label is automatically converted into an address, and the expression is evaluated further. The second operation is done in Assembly, Number, and possibly User windows. Assembly windows display the label and offset corresponding to the current instruction address on the left side of the screen and the labels and offsets indicating any effective addresses on the right side. Number windows suppose that the value typed was an address and try to find the corresponding label and offset, which are displayed in the bottom right corner of the window if they were found. User functions may also identify addresses using labels, but there are no examples of that in the standard user areas.

☞   *Evaluating a label* is converting it into an address. *Recognizing* an address is converting it into a label plus an offset.

☞   Each label is assigned an address and a *recognition range*. The recognition range is the range of memory, beginning with the label's address, any references to which shall be recognized as the given label plus an offset.

⇨   In general, no recognition range should be greater than $FFFF. Also, most of the label routines dealing with resources will either ignore or truncate resources which are longer than $FFFF bytes.

Labels are always displayed as eight characters, possibly padded on the right side with spaces if there are fewer than eight characters. They are displayed exactly the way they appear in the label table or code, which means that they are not converted to upper case for the purpose of displaying. Labels are entered into expressions by enclosing them in double quotes ("). They must not contain double quotes themselves. More than eight characters may be typed for a label, but all but the first eight are ignored. If fewer than eight are typed, the remaining characters are set to spaces. Unlike ASCII constants, labels typed into the Monitor *are* converted to upper case. The Monitor ignores case while searching for labels.

☞   Type labels by enclosing them in double quotes. Although labels are *displayed* in their original case, case is not important when comparing labels, and you may type labels in either lower or upper case. You may enter more than eight characters, but only the first eight are significant.

These names are used for the registers:

Variable	Value	Register name
A0 to A7	RA0 to RA7	Address registers.
D0 to D7	RD0 to RD7	Data registers.
SP*	SP	Same as A7 or RA7. *For anchoring windows only.
SSP*	SSP	System stack pointer. *For anchoring windows only.
USP	USP	User stack pointer. (Normally unused in the Macintosh)
PC	PC	Program counter.
SR		Status register.
CCR		Condition code register.
N	N	The result of the last Number calculation.
V	V	Result of Find, Heap, and other routines.
	USER	The beginning of the user area.
	DSPT	The address of the ROM A000 trap dispatcher.

Expressions can be made by combining numbers and register values using these binary operators:

+	Addition
–	Subtraction
*	Multiplication
/	Signed division
\	Signed modulo (The result has the same sign as the quotient would have.)
		Logical OR
^	Logical exclusive OR
&	Logical AND

The following unary operators are also allowed:

~	Logical NOT
@	Indirection (The four-byte value at the given memory location. The given memory location must be even; if it isn't, the menu bar will flash.)
+	Positive number
–	Negative number
!	Given an A000 trap number (modulo $200), return that trap's address

Consecutive unary operators are evaluated from right to left. Binary operators are evaluated according to this order of precedence:

*/\	Evaluated first
&	
+ –	
^	
		Evaluated last

Consecutive binary operators with the same order of precedence are evaluated from left to right. Triangular brackets (< and >) may be used to modify the order of evaluation. Depending on the complexity of the expression, between five and ten levels of parenthesis are allowed. All operators use 32-bit arithmetic.

Here are a few sample expressions:

0	Evaluates as 0.
A0	Evaluates as $A0 unless it is interpreted as a variable reference to address register 0 by the assembler.
$A0	Always evaluates as $A0.
2+2	Evaluates as 4.

Absolute labels also have an ending address to prevent cases such as addresses around $7F000 being identified using a label pointing to $400. The ending address specifies the end of the recognition range (it *is* included in the recognition range). Resource-relative labels do not have explicit ending addresses, but their recognition ranges end at the ends of their resources. If an address could be identified using more than one table label, the one higher in memory is used (for recognition purposes, of course. Obviously, either one may be used for evaluation. An example should make it clear: suppose that there are two labels, ALPHA at $20 to $300 and BETA at $40 to $274. Address $10 would not be recognized at all, and neither would $500. $3F would be recognized as "ALPHA   "+$001F, but $40 would be "BETA   "+$0000. Finally, $274, $275 and $300 would be recognized as "BETA   "+$0234, "ALPHA   "+$255, and "ALPHA   "+$2E0, respectively. On the other hand, nothing prevents you from *typing* "ALPHA"+254 to specify $274.

In recognizing labels, whenever there is a conflict between two or more *table* labels, the one highest in memory will be used. The results of a tie are unspecified.

Here is some more technical data on the storage format of both kinds of table labels. It is also helpful in understanding exactly what these labels can do. Each label record contains 16 bytes to make it easily readable in a Dump window. The bytes are assigned as follows:

Byte	0	1	2	3	4	5	6	7	8 to $F
Absolute	0	-begin	address-		0	-end	address-		label name
Resource-relative	---resource	type---			---ID---		-offset-		label name

Finally, there exist pseudo-labels which are not really labels but objects recognized by the address recognize routine. They are the A000 trap entry points and the DSPT variable. These items are recognized as if they were labels, but do not contain any quotes. To evaluate them, type !_routine name to find the address of the A000 trap routine and DSPT to find the address of the A000 trap dispatcher. Only ROM locations and system heap nonrelocatable blocks are searched while recognizing these pseudo-labels. To prevent excessive misidentification only the first $800 bytes after an A000 trap entry point are attributed to that label. Just like in table labels, the A000 trap whose entry point is highest in memory but not above the given address is attributed to that address.

One more issue remains to be resolved: what happens when several of the above types of labels may be used to *recognize* a particular address. The label highest on the order of recognition is given. The order of recognition is:

1. Table labels
2. Embedded name labels
3. Resource/ID labels
4. Pseudo-labels

Although this happens less frequently, it is possible for a given label to be *evaluated* in more than one way (For example, when there is a routine named "CODE0001". In this case the order of recognition is again followed, which means that in the example the routine would be given and not the CODE resource. If there are several embedded name routines with the same name, one of them will be picked, but I shall not make any guarantees as to which one. Under normal operation there cannot be two or more table labels with the same name, but that situation could arise if the label table is altered directly. Again, no guarantees will be made about which label will be used.

Since the labels depend on much of the system being in a consistent state, they may not always be reliable (although special precautions have been taken to avoid following things like odd or NIL pointers). In other cases it may be preferable to turn off one or more of the label types. Finally, some of the label recognition routines may not be particularly fast. Large Assembly windows with slow label routines tend to severely degrade the Monitor's performance. Try opening a large Assembly window to ROM and see how slowly the Monitor responds to typing and mouse clicking. On the bright side, the slowest label recognition routine, the pseudo-label recognition routine, is called only when disassembling ROM or system heap nonrelocatable blocks. Assembly windows pointing to places like CODE blocks run at a normal speed unless you have several hundred labels defined in the label table.

There are three basic kinds of labels plus two kinds of pseudo-labels. The labels may be either *embedded name labels*, *resourcelID labels*, or *table labels*. Table labels may further be subdivided into *absolute labels* and *resource-relative* labels. Here are the explanations of these types of labels:

Embedded name labels use names placed in code resources to identify code routines. The names are placed there by some Pascal compilers and may also be included in assembly language routines by using the .ASCII (or equivalent) assembler command. The specific method of recognizing embedded name labels is left to the user area. The default embedded name searching user routine searches for these labels *only* in CODE resources. A routine to which an embedded name label is assigned must begin with a LINK A6,#____ instruction and end with either RTS or JMP (A0). An UNLK A6 must be present within 20 bytes before the RTS or JMP (A0). The name of the routine must be placed immediately after the RTS or JMP (A0), and must be at least eight characters long. Moreover, it must consist of eight valid ASCII characters (ASCII values $20 to $7E), but the first byte may optionally have its 7th bit set. Finally, because of performance considerations there is a limit on the length of the routine: it should not be longer than about 4000 bytes. If at least one of the above conditions is not satisfied, the label is not recognized. If recognized, the label is set to the address of the LINK instruction, and the recognition range extends to the RTS or JMP (A0).

Note that the routine may have internal RTS or JMP (A0) instructions as long as there is no UNLK A6 preceding them within 20 bytes. There can be no internal LINK A6,#____ instructions, since any such instruction would be used as the beginning of the routine.

☞   If you are using a Pascal compiler to generate the embedded name labels, make sure that the necessary compiler options are turned on.

☞   Sometimes it is possible for the embedded name label routine to recognize spurious labels. This may happen if there are ASCII characters present after the end of a subroutine.

ResourcelID labels are a convenient way of identifying resources. A resource/ID label consists of the resource type in the first four characters and its ID in hexadecimal in the last four characters. Anything belonging to any resource is recognized using a resource/ID label containing the correct resource type and ID. For example, if you typed "DITL0080"+34 on the top line of a Dump window, the address of the window would be set to the $34th byte of a DITL resource with an ID of $80. If you open Assembly windows to code segments with no other labels in them, they will be most likely identified by using resource/ID labels.

There is one rare circumstance when this function might not behave as expected. The Monitor converts the label to upper case, and the label evaluate routine then compares the resource type against the first four characters of the label. This works fine except when the resource type contains lower case characters. Such resources cannot be specified by using labels in expressions. Recognition works fine for all resources, though.

☞   Using resource/ID labels is a quick way to open windows to specific resources, as long as the resources are present in memory. If not, use the user area LoadRes function. Also use LoadRes to find resources with types containing lower case letters.

Table labels are the labels you may add or modify. You may also load them from .Map files via a user area function (See Label File Load). You may add or remove table labels by using the Label Add/Remove user area function. Before you can do anything with table labels, however, you must allocate space for a label table by using the Label Table user function. As you can see, table labels are quite dependent on the user area, which also means that they can be changed in many aspects. The information given here applies to the predefined user areas only.

Table labels may be either absolute or resource-relative. Absolute labels refer to an absolute memory location, while resource-relative labels refer to memory locations within certain resource blocks such as code segments. Absolute labels are most useful for identifying low-memory variables and ROM addresses. Resource-relative labels are used to as an alternate way of identifying code routines, with the advantage that names do not have to be entered into the code and the disadvantage that the label table has to be allocated and loaded.

The Monitor's Functions

Dump

The top line of the Dump window serves to allow you to enter the beginning address of the window. When the window is first opened, the address is set to zero. You can set the address by typing a number or an expression there and pressing Return or Enter. The address is automatically aligned to a word boundary. If the cursor is already in a Dump window, a quick way to get to its top is to press Tab.

If you type Return on the top line without entering an address, the previous address of the Dump window is incremented by two, and the window is updated. This is useful for fine adjustments of the window.

It is also possible to "anchor" Dump and Assembly windows to particular 68000 or Monitor registers. It is done by entering the name (variable reference name) of the register in parenthesis on the top line of the window, optionally *preceded* by an offset. All data and address registers may be used, as may SP, SSP, USP, PC, N, and V. The V register is particularly useful in anchoring windows during searches. Examples of anchoring are: (A0), (D5),-20(PC), 8(V), and RA0(A0). In the last example the RA0 was a value reference, while the A0 was a register variable reference.

On the left side of the screen are the addresses of the sixteen bytes displayed on each line of the window. After the colons some symbols may be present; they are:

P	The address of the line is the same as the program counter.
S	The address of the line is the same as the system stack pointer.
U	The address of the line is the same as the user stack pointer.
0 to 6	The address of the line is the same as the value of that address register.
*	One of the breakpoints is set at the address of the line.
N	The address of the line is the same as the value of the last Number calculation.
V	The address of the line is the same as the value of V.

No more than two symbols are displayed per line, even if more than two are applicable.

To enter data into memory, move the cursor into the data section, and change the numbers there. Of course you may also enter expressions, but all values have to be between -$00 and $FF. When you are done changing your line, press Return. The cursor will then move to the first unchanged byte. Before you press Return, however, make sure that the cursor is still in the hexadecimal part of the dump.

You may also enter ASCII data by changing the ASCII part of the dump. Press Return or Enter to accept the new data, but make sure that before doing that the cursor is in the ASCII part of the dump. The cursor will then move to the ASCII value of the first unchanged byte.

☞   Do not enter the ASCII text in the ASCII portion of the window in single quotes. The text in ASCII portions of Dump windows is not converted to upper case even though it is not between quotes.

☞   In ASCII portions of Dump windows *only*, in order to avoid entering extra spaces at the end of the line, Return is made to behave like Enter in that it does not clear the rest of the line.

Enter may also be used to accept the new hexadecimal values. Be careful, however, because it will also accept the values left in the remainder of the field as if you typed these values yourself. This may cause problems like this one:

```
Before    00 11 22 33  44 55 66 77  88 99 AA BB  CC DD EE FF   .."3DUfw........
Changes   00 11 .17 .18 .19 2+2 'x' 88 99 AA BB  CC DD EE FF   .."3DUfw........
After     00 11 11 12  13 04 78 88  99 AA BB CC  DD EE FF FF   ......X.........
```

Incidentally, you can time the relative speeds of having various windows open on the screen by opening a small Dump window pointing to location $16A. Make sure that the cursor is *not* on the second line of the Dump window, and observe the changes in the value of $16D. Open other windows elsewhere on the screen and see how the increments in $16D get less frequent but also greater. Open a large Assembly window pointing to ROM, and you can get an idea how much it slows the Monitor.

For the reasons outlined in the above paragraphs you have the option of turning off parts of or the entire label recognition system in the Options window. See the description of that window for more information.

☞   Use the Options window to disable the label system. Do it to prevent crashes when opening Assembly or Number windows, speed up the monitor the Monitor's pace, or if you simply do not wish to use labels.

Exiting the Monitor

When you want to leave the Monitor and either launch an application, go into Configuration, or exit to the Finder, use the Exit function, either by typing ⌘E or clicking in the Exit area in the menu bar. The Main Menu will reappear. The Monitor will stay in memory and can be called by interrupt.

Reentering TMON

Even after you have exited to the Finder or launched another application, you may re-execute TMON either by clicking on its or any user area's icon, although if you use a user area, it will be ignored. Doing this will display the *second* Main Menu, from where you may either re-initialize the Monitor, enter Configuration and save the user area, launch another application, or simply quit again.

Permanently Leaving the Monitor

The Monitor continues existing in memory and can be called by pressing interrupt until you either turn off the Macintosh or press reset. There is no other way to dispose of the Monitor.

☞   Extensions are prohibited in instructions that have no size or only one size. This includes instructions such as MOVE D0,SR.

If the A000 trap names have been loaded, the A000 traps are displayed by their names. All trap names begin with an underscore. Unlike the 68000 mnemonics, the trap names contain both upper and lower case. In some cases the names are followed by a single hexadecimal digit; this occurs if bits 8 (9 for stack-based traps) through 11 of the trap word have been set in a nonstandard way. The standard values of bits 8/9 through 11 are:

1 for _GetZone, _MaxMem, _NewPtr, _NewHandle, _HandleZone, _RecoverHandle, _ReservMem, _GetTrapAddress, _PtrZone, _CompactMem, and _PurgeMem. (These are the eleven trap names that produce different values in expressions. See Numbers.)

8 for stack-based traps (trap with numbers greater than $50 with the exception of _UprString ($54) and _SetAppBase ($57)).

0 for the other traps.

You may also enter A000 trap names to be assembled. If you wish, you may enter the previously mentioned hexadecimal digits after the names. They may even be expressions as long as they evaluate between 0 and $F. Of course, you do not have to worry about upper and lower case when entering A000 trap names, since all input not between quotes is converted to upper case anyway.

There are two extra instructions which do not have Motorola's mnemonics. They are ???? and ROM. They both have one argument that is a number between 0 and $FFFF. They are mnemonics used for unimplemented instructions. The disassembler uses the ROM opcode for A000 traps which have no names if the names have been loaded or for all A000 traps if the names have not been loaded. ???? is used for all other unimplemented instructions. The assembler doesn't distinguish between the two opcodes. It also allows the use of these opcodes to assemble A000 traps even if the names of the corresponding A000 traps have been loaded.

Most addressing modes are standard. Numeric expressions may be used anywhere numbers are allowed except in the names of data and address registers. There are nevertheless several differences from the standard, and those are explained below.

The PC relative with offset and PC relative with index and offset addressing modes may each be specified in one of four ways: ^destination, *, *+offset, and *-offset for the PC relative with offset addressing mode and ^destination(Rn), *(Rn), *+offset(Rn), and *-offset(Rn) for the PC relative with index and offset mode. * stands for the current program counter value (the address of the first word of the instruction).

☞   The ^ symbol may be omitted if doing so would not make the instruction ambiguous. Specifically, it may be omitted in all branch and DBcc instructions. Omitting it in an operand of a MOVE instruction would probably cause the addressing mode to be interpreted as absolute or register indirect with offset.

☞   Keep in mind that labels may be used anywhere in expressions.

Be careful about using USP as an addressing mode. It may only be used as a variable in MOVE instructions between the user stack pointer and an address register. Since these instructions have only one size, long, a mnemonic extension is prohibited. An attempt to use USP in any other instruction, for instance MOVE USP,D0, will cause the USP to be interpreted as a value and evaluated as an absolute addressing mode whose address is the current value of the user stack pointer. See the Numbers section for an explanation. The same problem will occur if you try to use SSP or SP in an operand.

The register list after a MOVEM instruction is standard. The register may be listed in any order and separated by either dashes to indicate ranges or slashes to separate two registers. All these generate the same instruction:

```
MOVEM D0/D1/A3-A0/D6-D7,-(A7)
MOVEM D0-D1/D6-A3,-(A7)
MOVEM D0/D1/D6/D7/A0/A1/A2/A3,-(A7)
```

The best way to learn to enter data into a Dump window is to experiment, preferably on a block of unused memory. Locations between the top of the application heap (which may be found by looking at the top line of an application Heap window) and the bottom of the stack (which is pointed by register A7 in the Registers window) are usually unused. On a 512K Macintosh locations around $40000 are good in experimenting.

Dump windows may be scrolled and resized as described in the Windows section.

Do not enter addresses above ROM in the top line of the Dump window unless you are sure you know what you are doing. If you know the addresses and functions of the I/O devices, you may sometimes be able to change their settings directly from a Dump window. Remember that since the screen is constantly being refreshed, the addresses shown in the Dump window are constantly being read. This prevents the displaying of SCC registers, as even *reading* the wrong address will reinitialize the SCC, freezing the mouse (See Mouse Unfreeze in this chapter). VIA registers, however, may be displayed and even changed. Make sure that the byte you want to change is the first in the Dump window's line. Type a new hexadecimal value (or expression) over the old value and press Return (not Enter!). This way only the first byte of the line will be written. If you pressed Enter, all bytes on the line would be written. If you modified some byte other than the first and pressed Return, all bytes up to the one after which you pressed Return would be written. Both of these cases may cause some very undesirable effects.

Assembly

Assembly windows behave very similarly to Dump windows except that, of course, they contain assembly language data. The address of the window on the top line is set in the same way as in Dump windows. The symbols after the colon after the address of each line are also the same as in Dump windows.

As in a Dump window, typing Return on the top line without entering an address increments the previous address by two. Use this if the window appears to be misaligned in the instruction stream.

The Assembly windows attempts to use labels to recognize the address of the instruction and any effective address that the instruction may contain. The recognition of the instruction address is displayed just before the instruction itself, and may not be edited. If the instruction contains an absolute, relative with offset, or relative with index and offset addressing mode, the address specified by that mode is recognized on the right side of the screen in the form of a comment. If there is more than one such addressing mode present, only the *last* one is recognized. The recognition information will not be displayed if there is not enough room on the line.

Sometimes you may want to type an instruction on the line but cannot fit it due to the label on the left side of the screen. In that case, use the Options window to temporarily turn off labels, assemble your instruction, and turn the labels on again. Incidentally, the instruction that causes the longest disassembly is: MOVEM.L ^_____ (D0.L),D0/D1/D3/D4/D6/D7/A0/A1/A3/A4/A6/A7. (Note the missing registers in the register list. If D2, D5, A2, or A5 were present, the disassembler would use register ranges.)

Recognition of labels can be very convenient, but it may also slow the window. See the Labels section for information on the usage of labels and fixes for some problems.

To assemble instructions, type or change the instructions listed and press Return or Enter. If you entered an entirely new instruction, you will probably want to use Return. If you just changed an immediate value or an address in an existing instruction, it will probably be more advantageous to use Enter. Again, if you are unsure about the consequences, experiment on an unused area of memory.

No spaces are necessary before the mnemonics, but their presence there will not harm anything. At least one space is necessary between the mnemonic and the operands, if there are any. Spaces are *not* allowed anywhere else except at the end of the line and in ASCII constants and labels. The line may also have an optional comment at the end of the line. The comment must begin with a semicolon (;). The primary purpose for comments is to ignore the recognition information provided by the disassembler.

The standard Motorola mnemonics are used for the disassembler. In the assembler the use of the Q (quick), A (address), and I (immediate) suffixes after the mnemonic is optional. If a suffix is not included, the quick form of the instruction is going to be used whenever possible. The mnemonic may, in some cases, be followed by a .B, .W, .L, or .S extension. Extensions are prohibited in instructions that have either no or only one size.

Before	007000	007432	--------	012FFE	--------	--------	--------
Changes	7410 -	7422 (Return pressed here)					
After	007410	--------	007422	012FFE	--------	--------	--------

The breakpoints are implemented as TRAP #$F instructions. They are not put into memory until after the first instruction is executed after you leave the Monitor. This prevents you from having to remove a breakpoint in order to continue your program after it hit that breakpoint. Unfortunately this also has some side effects; see Trace Flag Side Effects for more information and some warnings. See also **Breakpoints** in the Exception Handling section.

⇨   You can cause many problems by setting breakpoints in sections of code that will be subsequently moved or written to the disk. The breakpoints will remain in the code, but they will be no longer recognized by the Monitor as breakpoints; instead, the Monitor will treat them as TRAP #$F instructions. Moreover, you will not be able to remove these breakpoints unless you know what instructions were "under" them before they were set.

⇨   The same problem as stated above may happen if you have breakpoints set and click the Monitor button in the Main Menu to re-initialize the Monitor. Fortunately, this should not happen often since there is little reason to set breakpoints in TMON code.

☞   The difference between a breakpoint and a TRAP #$F instruction is that after a breakpoint has been encountered the next instruction to be executed is the instruction "under" the breakpoint, while after a TRAP #$F instruction the next instruction to be executed is the next instruction in the code.

Registers

The Registers window contains saved values of all 68000 registers. All registers are displayed as 8-digit hexadecimal values except the status register, which is displayed in binary with the flags indicated as either upper or lower case letters. The interrupt mask in the status register is displayed as three binary digits. To change any of the registers move the cursor to the appropriate register, enter a new value, and press Return or Enter. See The Cursor and the Editing Facilities for an explanation. As in Breakpoints, more than one register can be changed at a time.

When changing the status register enter the name of the flag in lower case, a 0, or a period to clear a flag, or the name of the flag in upper case or a 1 to set the flag. Beware of changing the S flag. If its state is changed and Enter is pressed, in the next field the other stack pointer is going to be expected, causing the menu bar to blink. Therefore, it is best to press Return after changing the supervisor flag.

☞   A trick is used to disable the conversion of flags to upper case. The entire status register is enclosed in quotes, allowing you to use upper case to set flags and lower case to clear them.

The Monitor won't allow odd values in the program counter or either stack pointer.

Heap

The Heap window displays the contents of a heap zone. When it is first opened, it displays the application heap zone, but it may be toggled between the application and system heap zones by pressing Return or Enter with the cursor on the top line. The top line displays, in addition to the name of the current heap zone, its location and number of free bytes, both in hexadecimal. The addresses of the zones are stored in location $2A6 for the system zone and $2AA for the heap zone. These two locations are maintained by the Macintosh operating system; you should normally not be changing them.

As in the second example, if a range contains both an address and a data register, the address registers are considered to "follow" the data registers. You may not enter a list that doesn't contain any registers.

Here is a summary of the addressing modes:

Dn	Data register direct
An	Address register direct
(An)	Register indirect
(An)+	Postincrement register indirect
-(An)	Predecrement register indirect
offset(An)	Register indirect with offset
offset(An,Rns)	Register indirect with offset and index
address	Absolute
^*address*, *, *+*offset*, or *-*offset*	Relative with offset
^*address*(Rns), *(Rns), *+*offset*(Rns), or *+*offset*(Rns)	Relative with offset and index
#*number*	Immediate
USP, SR, CCR	Implied register

Rns is an abbreviation for either a data or an address register optionally followed by either a word or longword size indication. Some examples of Rns are D0, A0, D3.W, and A7.L.

Next is a table of the possible ranges of numbers. For the -128 to 255 and -32768 to 65535 ranges the values above 127 and 32767, respectively, are just positive equivalents of the negative values.

1 to 8	$00000001 to $00000008	Values in ADDQ, SUBQ.
0 to 15	$00000000 to $0000000F	TRAP value and value after A000 trap name.
-128 to 127	$FFFFFF80 to $0000007F	MOVEQ value.
-128 to 255	$FFFFFF80 to $000000FF	Byte immediate values, offset in register indirect with offset and index addressing mode.
-32768 to 32767	$FFFF8000 to $00007FFF	Absolute short addressing mode.
-32768 to 65535	$FFFF8000 to $0000FFFF	Word immediate values, offset in register indirect with offset addressing mode.
0 to 65535	$00000000 to $0000FFFF	???? instruction.
all values.	$00000000 to $FFFFFFFF	Absolute long addressing mode and long immediate
*-126 to *+129		Offset in relative with index and offset addressing mode.
*-126 to *+129 excluding *+2		Offset in a short branch.
*-32766 to *+32769		Offset in a relative with index addressing mode or a long branch.

During reverse scrolling the disassembler tries to find the preceding instruction, but that is not always possible. To make a guess at the length of the preceding instruction it goes up to 128 words back in memory. If it finds that no preceding instruction exists, it goes back one word and disassembles from that location. If more than one preceding instruction is possible, it chooses one of them. In both cases it will refresh the entire window, significantly slowing the reverse disassembly.

Breakpoints

Only one Breakpoints window can be open at a time. It contains up to seven breakpoints. To enter or change an address of a breakpoint move the cursor to the appropriate place on the lower line, enter the new address of that breakpoint, and press Return. Avoid having two breakpoints at the same address. To delete a breakpoint do the same thing except instead of an address type a dash. You may also change more than one breakpoint at a time by separating the addresses and/or dashes with spaces. For instance:

In addition to the items listed above, the components of WmgrPort and all windows on the window list are identified. These components are identified first by either by the phrase (Window @$..), which indicates the location of the window to which they belong, or (WmgrPort), indicating that they belong to the WmgrPort. After one of these two identifications one of the following messages is displayed:

VisRgn	The region of the window which is visible on the screen.
ClipRgn	The clipping region.
PicSave	Data for a picture being saved.
RgnSave	Data for a region being saved.
PolySave	Data for a polygon being saved.
StrucRgn	Structure region of window.
ContRgn	Content region of window.
UpdateRgn	Update region of window.
WData	Window-defined data.
WTitle	The title string.
WPic	Window's picture used for updating.
DlgItemList	Item list (dialog windows only).
TEHandle	TextEdit record (dialog windows only).
Item #$.. type $..	An item in the window's DlgItemList (dialog windows only). The first number is the item number (first item is 0, second 1, etc.), and the second number is the item type.
Control	Any control belonging to the window. This is displayed only if the control could not be identified as an item in that window's dialog list.

☞   If the Heap window crashes, "hangs", or causes the screen to flicker when opened or scrolled, the identification routines are getting lost, and you should turn them off. You also may want to turn them off for other reasons; for example, it is possible that identification might slow the Monitor too much. That is most likely to happen if you write your own heap identification routines, and do it inefficiently; the standard identification routine is quite fast. Anyway, if for any reason you want to turn off either identification of non-resource objects or identification of resources, use the Options window.

A Heap window may be scrolled up or down by using the arrows on the right side of the window. There are two quick ways to scroll the window up to the top of the heap: either close the window and then reopen it or type Tab, Return, Tab, Return. This will toggle the window between the two heaps and back, with the side effect that at each toggle the window will display the top portion of the heap. Since there is no quick way to scroll the window to the bottom of the heap, it is recommended that Heap windows be left open as long as they are needed. No harm will be done if the window is left open while the information in the heap changes; the window will just update itself. It is sometimes interesting to watch an open Heap window while the Scramble now user area function is repeatedly executed.

☞   More than one Heap window may be opened by using the Shift key. Two or more windows may even be opened to the same heap zone without harmful effects.

☞   Sometimes it is possible for a Heap window to turn completely blank except for the top line. This will happen if the window was initially displaying an area near the bottom of the heap and then the heap contracted. In that case you can either do nothing, scroll the window up until you see the heap blocks, or follow one of the procedures described above for quickly moving the heap window to the top of the heap. For a demonstration of this phenomenon move the Heap window to the bottom of the application heap and then use the MemHeap option in the user area, causing some blocks to be purged.

You cannot enter any information into Heap windows. Any changes to the heap structure have to be done by opening Dump windows to the appropriate places and changing data in them. A Heap window may then be used to verify that the changes were done correctly.

The Monitor will crash if the contents of either of these locations ($2A6 or $2AA) is odd or does not point to RAM.

The Heap window displays the position, length, and other data for each block in the heap zone. For each block the following information is displayed:

An asterisk if the block is immovable or a space otherwise.

The address of the beginning of the block in hexadecimal. Memory Manager's block header for this block is stored in the 8 *preceding* memory locations.

The logical length of the block in hexadecimal.

The size correction as a single hexadecimal digit. The size correction is the difference between the logical length and physical length-8. The physical length is the difference between the address of the next heap block and the address of this one. The size correction represents the number of unused bytes in this heap block.

One of the following phrases:

Free	a free block.
Nonrel	a nonrelocatable block.
Handle at $.. (LPR)	a relocatable block. The address of the handle is given. The three letters in parenthesis are the flags found in the high byte of the handle, set if they are in upper case or clear if in lower. L is set if the block is locked, P is set if the block is purgeable, and R is set if the block is a resource.
INVALID	any block that is not consistent. See the comment on reliability below.

Nonrelocatable and relocatable blocks may then contain further information helpful in identifying them. All such information except the resource file information is generated by a user routine, and is therefore subject to change and customization. If you find the identification of blocks inadequate, you are welcome to try to change it to suit your needs. Information on doing this is listed in the Creating Your Own User Functions chapter.

All relocatable blocks with the R flag set are checked against the list of open resource files and their resources. If they are present, their resource file number, type (ASCII letters in quotes), and ID number are displayed. All other blocks are passed to the user routine which should identify as many of them as possible. The default routine identifies the following:

GrayRgn	$9EE	The rounded region defining the desktop.
MenuList	$A1C	The current menu bar list.
TEScrap	$AB4	TextEdit scrap.
Scrap	$964	Memory scrap.
SaveVisRgn	$9F2	A region used by the Window Manager.
WmgrPort	$9DE	A grafPort used by the Window Manager.
FinderInfo	CurrentA5+$10	The Finder information handle (in system heap).
ParamText0	$AA0	The first parameter in the last ParamText call.
ParamText1	$AA4	The second parameter in the last ParamText call.
ParamText2	$AA8	The third parameter in the last ParamText call.
ParamText3	$AAC	The fourth parameter in the last ParamText call.
Resource map $..		Resource map of the given resource file.
Window #$.., kind $..		A window found by following the window list. The first number is the number of the window (0 is the frontmost window, 1 the next one, etc). The second number is the value of windowKind for that window.

(The hexadecimal numbers are either handles or pointers to the items above.)

. Local reference	R System reference
. Load into application heap	H Load into system heap
. Not purgeable	P Purgeable
. Not locked	L Locked
. Not protected	T Protected
. Not preloaded	1 Preloaded
. Do not write into resource file	W Write into resource file
. U flag clear	U U flag set

After the flags either the memory location of the resource is displayed or nowhere if the resource is not in memory. Afterwards, if the resource is a system reference, the phrase Sys ID= is displayed along with the system reference number. If there is a system reference name, it is also displayed in single quotes, preceded by the words sys name. The name of the resource (also in quotes) is displayed last if it is present. Note that if two names are displayed, the first one is a system reference name and the second one is the resource name. These cases are so rare, however, that for all practical purposes you should not be concerned with them; should one arise, you can refer to this section of the manual. The system file usually contains a few resources with names.

File windows are very similar to Heap windows in the aspects of scrolling. In fact, the last few paragraphs of the description of Heap windows apply to File windows as well with the difference that File windows do not use any identification user routines. There are two more minor differences: in order to quickly move a File window to the top of the file, instead of typing Tab, Return, Tab, Return, type Tab and then Enter. This will re-enter the current file number into the top line of the window, also causing the window to be moved to the top of the file. Finally, when browsing through the file using a Dump or Assembly window anchored to V, you will find that blocks which have not been loaded into memory are ignored. Pressing Return or Enter on a line containing one will not change the value of V.

Just as in the Heap windows, any information displayed in the File windows is checked for accuracy. NIL or odd pointers will not cause the File windows to crash, although there may be more elaborate ways of making them crash. Any invalid files are simply not displayed. Any invalid resources are displayed as nowhere.

Exit, GoSub, Step, and Trace

These four functions all leave the Monitor, starting execution at the current PC. They differ in the duration before they return to the Monitor. Trace returns to the Monitor as soon as the next instruction is executed; Step is just like Trace except that it treats A000 traps as units; if the next instruction is an A000 trap, it is allowed to complete, and only then does control return to the Monitor. GoSub also like Trace, except that both A000 traps and JSR or BSR instructions are treated as units. GoSub is a quick way of skipping subroutines and executing just the main body of the program or a procedure. Finally, Exit leaves the Monitor indefinitely; that is, until the next exception.

All four functions restore all registers, the screen, and the cursor before they leave the Monitor. They all do not put breakpoints into memory until the second 68000 instruction executed to allow continuing after a breakpoint has been hit.

When the Monitor is reentered after GoSub, Step, or Trace, one of two messages may appear near the top of the screen. "Trace interrupt at $_____" will usually appear; if, however, an entire subroutine or A000 trap was executed, "The A000 trap or subroutine has returned" will appear instead.

It is possible for the Monitor to refuse to proceed, displaying the interesting message, "I don't want to execute the next instruction." This will usually happen when continuing would not normally be to your best advantage. Specifically, the Monitor will refuse to continue if the next instruction is _Debugger or _SysError. (Exception to the exception: _SysError will be executed if the word value of D0 is 30 or 31.) Also, for GoSub and Step only, _LoadRes, _Launch, _Chain, and any A000 traps with both the stack-based and auto-pop bits set cause the Monitor to refuse to execute the next instruction. These latter A000 traps are restricted for GoSub and Step because these functions do not know where these traps return; unlike other A000 traps, these traps do not resume execution at the instruction following the trap.

There is a quick way of scanning the heap on either a Dump or Assembly window. Open one of these windows and anchor it to V (0 (V)). Then place the cursor in a Heap window on the line of the block at which you would like to look and press Return or Enter. That action will cause V to be set to the address of the block at which the cursor was located and *then* the cursor to be moved to the next block of the Heap window. By repeatedly pressing Return or Enter the entire heap may be scanned without having to type the addresses of the blocks. Remember, however, that the cursor in the Heap window will always be on the line after the line containing the block descriptor for the block currently being displayed in the Dump or Assembly window.

Finally, here is some information about error checking and reliability: aside from the values of locations $2A6 and $2AA the Heap window does *complete* error checking of the heap blocks it displays. When addresses of handles are displayed, the handles are checked to make sure that they point to the block. All pointers are checked to make sure that they are even and non-NIL and that they point to real memory. Any odd physical block lengths are rejected as are blocks with unusually large sizes. The heap zone pointers in the block headers of relocatable blocks have to point to the heap zone which is currently-displayed. Any blocks which do not pass the error checking are displayed as INVALID. You may use a Dump window to find why they are considered invalid.

Only one area may cause problems with reliability of Heap windows. In order to make the speed of these windows acceptable, the resource file maps are not fully checked when relocatable heap blocks are being identified. Should a damaged map claim that it has 65,535 resources, all will be examined in search of the one that is to be identified. Nevertheless, as with the heap zones themselves, all applicable pointers and handles in the resource file maps are rejected if they are NIL or odd.

The supplied user routine does error checking on the parts of the structures it examines. In this case error checking means that all pointers are checked if they are NIL or odd. The user routine will not stray on invalid data structures for too long. However, should you still encounter problems with the identification routine, you may turn it off by using the Options window. One indication of problems is the screen becoming fuzzy when a Heap window is opened; this is caused by some memory accesses above $40FFFF.

Blocks which are displayed valid in the Heap window are usually valid. There is one important exception: if, for one reason or another, some of the handles which are traced by the Heap function have been disposed but not set to NIL, they may still point to master pointer blocks. If another handle is allocated using one of the master pointers, it will be erroneously identified as the heap object to which that master pointer used to belong. This problem will usually be encountered if not all managers have been initialized. In this case some heap blocks may be incorrectly identified.

File

A File window displays the contents of any open resource file. When a File window is first opened, the numbers of all open resource files are displayed on the second line of the window. You may then enter the resource number of the file you wish to examine on the top line. File $2 is usually the system file and $20 the application file. If, after you are done examining one file, you wish to switch to another file, you may either enter the number of that file on the top line of the window or press Return there, causing a listing of the numbers of all open resource files to reappear.

Once you have entered a number of an open resource file, the contents of the file's resource map are displayed in the window. The top line of the window contains the address of the resource map as well as three flags which apply to the entire map. These three flags are stored in the 22nd byte of the resource map and are displayed as follows:

r	The map is read-write	R	The map is read-only
c	No compaction necessary	C	The map will be compacted
w	The map was not changed	W	The map was changed and will be written to disk

For each resource its type, ID number, flags, location, and optionally name and system reference data are displayed. The type is displayed first, in ASCII between single quotes, followed by the resource ID displayed as a four-digit hexadecimal number. The flags follow. They are displayed as upper case letters if they are set or periods if they are clear. The following abbreviations are used for the flags:

Number

The Number window asks you for a number or an expression, evaluates it, and displays it in hexadecimal, 32 and 16 bit decimal, ASCII, as an A000 trap name, and as a recognized address. It also sets the N variable to the value entered. The second line of the window contains, in order from left to right, N= (if the value displayed is equal to the current value of N), the value in hexadecimal, the value in signed 32-bit decimal, in parentheses the lower 16 bits of the value in decimal, and in quotes the value in ASCII. The A000 trap name corresponding to the lower 9 bits of the value is next, followed by the label recognition information that would be generated if the value typed were an address. If the A000 trap names have not been loaded, or if the particular trap has no name, the number is displayed instead. If there is no recognition information available for the particular address in the window, the recognition field is left blank.

More than one Number window may be open on the screen at a time. Several Number windows can serve as temporary memories to remember values which you would otherwise have to write on paper.

The N= indicator is not as extraneous as it may initially seem. Try opening two Number windows and entering different numbers into them. If no Number window contains the N= indicator and you want to see the current value of N, enter N into any Number window.

☞ Be careful about typing values such as $A122 (_NewHandle) into the Number window and expecting to see the right trap name on the bottom line. Since the A000 trap name routine uses the low 9 bits of the trap number, it thinks that you want trap $122 ($A922), which is _BeginUpdate. This problem occurs only with eleven A000 traps which have bit 8 set as an option. It is addressed further in the sections about Numbers and Assembly windows.

User

The user area is one of the most powerful features of this Monitor. It allows you to create your own functions or use the predefined ones. This section is just an overview of using the functions; information on creating new ones and an explanation of the predefined ones are included in a later chapter.

The top line of the user area contains the beginning address of the user area, which is also the value of the predefined variable USER. Afterwards the physical and logical sizes of the user area are listed. The physical size is set at boot time and cannot be changed. It indicates how much memory is reserved for a user area. The logical size is used by the Configuration functions only and indicates how much disk space the user area would take if it were saved. The logical size may be changed by typing a new value in its place and pressing Return or Enter. It must, however, be nonzero, a multiple of $100, and not greater than $6000. More information on logical and physical sizes is in the Configuration section.

To use a function in the user area, move the cursor to the appropriate line. Type as many parameters after the colon as the function requires (some require none), and press Return. The function will be executed and the cursor will return to the position after the colon. The function may return results in other places on the line, usually within the curly ({ and }) brackets. Some functions also put numbers after the colon so that you can just press Enter to execute the function again. Still others affect Monitor's registers, usually V. Functions can be created, like ShowScrn in the predefined area, that leave the Monitor altogether, while others may just provide additional parameters to be used by other User functions.

Most functions within the predefined area give a listing of the parameters they expect within parenthesis before the colon. If there is no such listing, the function probably doesn't need any parameters and is executed just by pressing Return. If the wrong number of parameters is supplied, the menu bar will flash.

Sometimes the function name itself is displayed in parentheses. That means that several functions have been encoded on a single line, and you can cycle through them by pressing Return immediately after the colon. A good example of such a function is Print (which should not be confused with the Print menu command).

☞ Use either Exit with a breakpoint or Trace if you want to continue execution without losing control after hitting a _LoadRes, _Launch, or _Chain. Traps with auto-pop bits set should only present a problem in older code; the auto-pop bit is not used any more. If you do find one, examine the stack to find the return address, set a breakpoint there, and use Exit.

⇨ Due to the way these routines function internally, be careful if the next instruction is MOVE SR, *dest*. The trace bit will be set in the saved value of SR, and you should clear it before continuing. This problem arises because these exit functions, including Exit, use the trace flag to single-step the next instructions, and only then do they put in the breakpoints and perform other tasks like reenter the Monitor.

☞ Even though the Monitor uses the trace flag for its internal stepping purposes, it will work correctly if you set it in the Registers window! A trace interrupt will be generated after every instruction, even if you use Exit. Keep in mind, however, that the trace flag is cleared by any A000 traps encountered.

Although GoSub and Step are quite clever, you should be aware of some of the problems they may cause. In order to obtain control after the subroutine or A000 trap returns, they save the address of the next instruction in the Monitor's variables and pass a dummy address pointing to a Monitor routine to the trap or subroutine about to be executed. Obviously, if that subroutine examines or changes the return address it got on the stack, this scheme will fail. You will have to avoid skipping through subroutines or A000 traps that examine or change the return address. A few A000 trap routines examine the return address and execute patches if they were called from a particular place in ROM; this is used as a method of fixing ROM bugs. This is a mild version of the problem described above, as it will cause trouble only if you use GoSub or Step on the A000 trap in the ROM place to which the patch is attached; the patch will not be executed.

Another potential area of problems is interrupting the subroutine or A000 trap that was called by GoSub or Step. The interrupt could be a press of the interrupt button as well as a breakpoint, illegal instruction, address error, user area A000 trap exception, or another exception. Whatever the cause, after that interrupt you may wish to step through your code at the new PC and use GoSub or Step again, on another subroutine. *This will work correctly* up to *eight* levels of recursion! Also, nothing harmful will happen if one of the levels of recursion is never completed. The Monitor can keep track of up to eight pending subroutine or A000 trap returns, and it will halt execution as it encounters each one.

Finally, in a case similar to the above one, suppose that the subroutine or A000 trap called by GoSub or Step is interrupted. After the interruption you choose to single-step through the rest of the subroutine, including the final RTS, JMP (A0), (or whatever) statement. Instead of getting the expected "Trace interrupt at $_____" message you will instead get "The A000 trap or subroutine has returned," but everything else will work as expected. The subroutine's return address will be removed from the Monitor's eight-entry list of return addresses mentioned above. If you wish to examine it, the eight-entry list is stored at USER-$2F0 [longword array], and the order of priorities of assigning the next entry is at USER-$2F8 [byte array].

Options

The Options window lets you enable/disable seven features of the Monitor. You may want to disable them because they crash, take too much time, or don't do anything useful. Whatever the reason, you may disable and reenable any of the features by moving the cursor to the correct line and pressing Return or Enter.

Master switch disables all label recognition.

Scan resources disables the scanning of resource files by the label routines. This turns off resource/ID resources completely. Since there is now no way of distinguishing CODE segments, the embedded name labels are also inoperative. Resource-relative table labels can no longer be evaluated or recognized.

Scan label table disables the user area label table routines.

Scan for names in code disables the user area embedded name label routines.

Identify A000 traps disables the pseudo-label Monitor routine.

Scan resources disables the identification of resource types, IDs, and files in Heap windows.

Identify items disables the identification of other Heap window items via a user routine.

Exception Handling

Overview

The Monitor intercepts all interrupt vectors except the ones essential to the functioning of the Macintosh. For extra security the Monitor's vectors are stored again into low memory every time the Monitor is entered unless a Configuration option is changed (See Configuration). The Monitor also has a self-check feature that will tell you if the Monitor has become unreliable. If it detects an error, it will reset the entire Monitor and display a message. You may then take an appropriate course of action. Finally, the user area functions may themselves generate exceptions that are intercepted by the Monitor.

Normal Exception Messages

When an exception is intercepted by the Monitor, the registers are saved and breakpoints removed. Then the current screen, cursor, and cursor position are saved and possibly compressed, the hardware is switched to display the main screen page if the alternate one was used, the Monitor's window and background are displayed along with a message explaining why the Monitor was entered, the user area's initialization routine is executed if present, and control of the computer is turned over to the Monitor. This section deals primarily with the messages that appear at this time.

The messages for most exceptions are self-explanatory. There is one thing, however, that may require an explanation. When the message gives the current value of the program counter, it sometimes states that the exception happened *before* that value and sometimes *at* that value. The program counter reported is always the value saved by the 68000 upon handling the exception. The 68000, however, sometimes saves the address of the next instruction and at other times the current instruction. This is the reason for the difference.

Address and Bus Errors

The messages for address and bus errors give more data because more data is saved by the processor. The program counter is given along with the address that was accessed when the error happened. The program counter value most likely isn't the address of the instruction that caused the error; in most cases it is somewhere nearby that instruction, although branch and jump instructions may affect the saved value of the program counter. The function code is displayed in parenthesis after the access address and tells what type of access that took place:

Message displayed	Function code saved by the 68000
user data	001
user program	010
supervisor data	101
supervisor program	110
exception	111
illegal	000, 011, or 100

The next field, instruction, tells whether the processor was executing an instruction at the time of the error. The only time it is *not* executing an instruction is during fetching an exception vector. The last field, mode, tells whether the access was a read or a write.

Breakpoints

The breakpoints are TRAP #$F instructions. TMON distinguishes them from true TRAP instructions by checking the program counter against its list of breakpoints. If it is present in that list, a breakpoint message is given; otherwise, a trap message is given. See the Breakpoints section in the Monitor's Functions chapter for more information.

Print

This function sends the contents of the topmost window to a serial port. See the Configuration section for more information about which port is used and how to change data like the baud rate. You may stop the printing by pressing the mouse button. If a handshaking protocol is active, you may have to hold the mouse button for a long period of time (possibly 20 seconds or more), because the state of the mouse button isn't checked while the Macintosh is waiting for the other RS232 device to allow it to send another character. Also note that since the mouse click is executed, if you tell it to print by clicking the mouse button over the Print area in the menu bar, you will give the Monitor another command to print the topmost window, which is just what you wanted to avoid! If an error occurs while printing, its error code will be stored on the Print line in the user area if you are using one of the predefined user areas. An error code of 1 will be stored if the printing is interrupted by the mouse button. An error code of 2 is displayed after an attempt to use the user area's Print file function on a nonexistent resource file. See also the explanation of the Print function in the user area and the Printing Problems section for some of the possible communication problems.

Never use ⌘-interrupt to stop a long printout; press the mouse button instead. If you do not heed this advice, the Monitor may behave in ways *stranger than could be imagined*.

Mouse Unfreeze

This is the only function that does not appear in the menu at the top of the screen. It can be executed from the keyboard only by holding down the ⌘ key while typing M. This function unfreezes the mouse and at the same time turns off both serial ports. The mouse is probably frozen if it appears on the screen but doesn't move or doesn't appear on the screen at all. The Monitor must respond to keyboard commands if you want to use the unfreeze. The main cause of mouse freezing is accessing memory locations above $40FFFF. Programs that crash or otherwise go out of control frequently do that. Accesses to such memory locations tend to reset the SCC chip, turning off the mouse interrupts. This option will turn those interrupts back on.

If you should happen to freeze the mouse by opening a Dump or Assembly window with an address in that memory range, *first* move the cursor to the top of that window by typing Tab or using ⌘D or ⌘A, then change the address of that window to zero and press Return, and finally use the Mouse Unfreeze function. In File windows do the same thing but press Return alone, causing just the list of file numbers to be displayed. If using the unfreeze now doesn't help, follow the procedure for Heap windows.

If a Heap window causes the mouse freezing or if any of the above procedures doesn't work, you have to close the offending window. That requires you to move the mouse to the close box. In most cases that can be done by moving the mouse while repeatedly typing ⌘M. The mouse will move in small jumps, but with luck you will be able to close the window. If even this doesn't work, hold down the ⌘ key and press the interrupt button. Afterwards you can use the unfreeze function. This will reset the monitor, possibly damaging it, but now at least you can use it.

If this option was performed successfully, a message window will appear near the top of the screen informing you that the mouse has been unfrozen and that you should not try to use the serial ports. You may, however, use the Monitor's Print functions.

Possible Problem Areas

Mouse Freezing

See the Mouse Unfreeze section in the The Monitor's Functions chapter.

Interrupting the Vertical Retrace

When you press the interrupt button to enter the Monitor, there is a small chance that you will interrupt the vertical retrace handler routine. This could have a number of effects, ranging from none to a system crash when you leave the Monitor. Although this possibility can't be eliminated, its probability can be significantly decreased by making your program's vertical retrace queue short if it has any and by not moving the mouse when you press the interrupt button.

Can't Regain Control of the Monitor

If interrupt does not seem to work, hold down ⌘ and press interrupt. If you still have no effect, hold down both ⌘ and Option and press interrupt again. Should this still be unsuccessful after a few tries, you probably will not be able to regain control of the Monitor and should give up and press reset. See the Interrupt Button section for more information on this topic.

Trace Flag On

If you keep getting a trace interrupt after leaving the Monitor, you are either single-stepping or the trace flag in the Status Register is set. Clear it. Rember to clear the trace flag in the saved SR if you use Exit, GoSub, Step, or Trace when the next instruction is MOVE SR, *dest*.

Windows Crash or Are Too Slow

If the Assembly, Number, or User windows crash when opened or scrolled, turn off the labels using the Options window. If you can isolate the label routine that is causing the crashes, you may turn if off and leave the others on. If the Heap windows crash, you will have to turn off one or both of the options in the Options window dealing with Heap windows.

Follow the above procedure if you believe that Assembly or Heap windows are too slow to be convenient. Remember that a large slow window *visible* on the screen will slow all of the Monitor's operations, not just the ones dealing with that window. Covering such windows or decreasing their sizes may help.

Printing Problems

If you encounter problems with printing, make sure that the information in the Communications menu of Configuration is correct. It is also possible that the Monitor may not print correctly if some other program has opened and used the serial port designated for printing from the Monitor. If this is the case, try using the other port for doing printing from the Monitor. Also, since printing uses the Memory Manager, it may not work if the heap is in an inconsistent state.

Not Enough Memory to Launch Application .

Use the Screen Buffer option in Configuration to decrease the size of the saved screen. Some other suggestions include using a smaller user area and removing the A000 trap names.

System Error

The system error vector is also intercepted by the Monitor. All system errors except 30 and 31 will cause the Monitor to be entered with an appropriate message.

Interrupt Button

The interrupt button on the left side of the Macintosh generates interrupt exceptions with priority levels between 4 and 6. If the Monitor isn't currently executing, it will be started and an appropriate message will appear at the top of the screen. If it is currently executing, the interrupt will be ignored.

Sometimes pressing the interrupt button may have no effect. If the processor's interrupt level is 7, you can't do anything about it. If, on the other hand, something goes wrong with the Monitor and you desperately want to regain control, hold down the ⌘ key while pressing the interrupt button. This will re-initialize the Monitor to its original state, but that action will also erase some of the Monitor's variables, causing a message stating that the Monitor has been damaged to appear on the screen. Do not use Exit or GoSub after you have pressed ⌘-interrupt unless you initialize the program counter, stack pointer, status register, and any other necessary registers.

> The ⌘-interrupt action is one of last resort and should not be used often, as it may cause unpredictable damage to the Monitor.

> If you have reason to believe that the user area is damaged (⌘-interrupt causes the Monitor to hang), you can try to take a still more drastic course of action. Hold down both ⌘ and Option and press interrupt. This action, called Option-⌘-interrupt, will do everything ⌘-interrupt did plus erase the first 48 bytes of the user area, effectively clearing it.

Self-Check

Whenever there is a window on the screen the Monitor does a self-check. If the self-check detects some signs of damage to the Monitor's code or a stack overflow, it will re-initialize the Monitor, just as if you pressed ⌘-interrupt (see the previous section). There are three things that could cause the Monitor to display a message informing you that it has been damaged:

One of the ways the Monitor could be damaged is if its code has been modified. If that happens, you will get only a message stating that the Monitor has been damaged. In this case some function of the Monitor will no longer work because its code has been changed.

The Monitor also checks to make sure that its stack stays within the area of memory designated to it. If it overflows, you will get a message stating that the Monitor has been damaged.
In this case some user functions will no longer work because the stack is located immediately after the user area.

The third way the Monitor could be damaged is if an exception occurs while the Monitor is executing. The Monitor is executing anytime the Monitor's screen is visible. This includes execution of a user function by pressing Return in the user window. This does *not* include execution of user routines by calling them from an outside program. If this type of an error occurs, you will get two messages, one stating the nature of the exception and the other stating that the Monitor has been damaged. The moral of this is to make sure that your user routines are "safe" by using code to prevent exceptions such as address errors from happening in them.

User Exceptions

The user routines that are called from an outside program may enter the Monitor by executing TRAP #$F instructions from within the user area, displaying the message "User trap:" followed by text from the user area. Details on this are included in the User Routines Entering the Monitor section. To see an example of this use the ShowScrn function in the standard user area.

Alternate Screen Page Handling

If, when you press the interrupt button, the second screen page is shown, the screen will be switched to display the primary page and restored to the alternate page after you leave the Monitor. (User area functions ShowScrn and LoadRes do *leave* the Monitor, but they return to it later.)

⌘-Shift-1 to ⌘-Shift-4 Usage in the Monitor

If you attempt to use these functions while the Monitor is executing, no action will seem to take place until you exit the Monitor. Actually, these keystrokes will be saved and executed only after you exit the Monitor and the application program which you run after leaving the Monitor calls some A000 routines.

Crash while Exiting to Finder

On 128K Macintoshes you must use the **Configuration** options to make the size of the Monitor less than 26000 bytes if you want to be able to run the Finder. This means that if you want to use the otherwise standard user area, you must reduce the **Screen Buffer size** to 4000 bytes and use A000 trap numbers or, if you want to use A000 trap names, you must set the **Screen Buffer size** to 0.

Debugging Existing Applications

Using the Monitor with existing applications presents an array of problems. The Monitor is, nevertheless, flexible enough to allow it to be used with almost any program written for the Macintosh. That does not mean that you will not have to make adjustments. Some common changes which have to be made with some applications are disabling the Vector Refresh option, making the screen saving area smaller if memory is a problem, and loading the Monitor into the system heap instead of high memory. The first of the actions listed should be done if you encounter TRAP exceptions; the last if you find that the program you are debugging uses the alternate graphics or sound page.

The Monitor particularly dislikes changing the interrupt vectors and a few of the A000 trap vectors. Avoid changing the vectors _SysError and _PostEvent. If _SysError was changed, the Monitor will put it back to its previous address as soon as it regains control unless Vector **Refresh** was disabled. Do not change _PostEvent in such a way that events such as key down, mouse down, and mouse up are not posted; if you do, the Monitor will fail.

The ability of loading the Monitor into the system heap is a controversial one. If all Macintosh programs were well-written, they should not be affected by having a Monitor in the system heap. It appears, however, that some bugs in programs appear only if the system heap exceeds a certain size ($10000 to be exact). For that reason the option of loading the Monitor into high memory was provided. An example of a bug that will appear only if the Monitor is in the system heap is in an application that calls _SetTrapAddress to a routine in the application heap. That shouldn't be done, but if it is, it will cause no problems unless the routine is at or above location $10B00. You, of course, will avoid all such errors, and might even find it constructive to load the Monitor in to the system heap to make sure that none are present.

In some cases you still might have little idea why the Monitor can't be used with some programs. In that case, use **Trap intercept** to stop the program at the beginning (intercepting _InitGraf works well), and use **GoSub** to trace the program execution. If you find that the program crashes in one subroutine, you now know where that subroutine is and can try the process again except that the next time step through that subroutine. You can then use this divide-and-conquer approach to quickly find the instruction that gives the Monitor indigestion. Once you know what is wrong, you may be able to bypass or fix it.

Using RAM Disks and Other High-Memory Drivers

TMON respects the space reserved for RAM disks and other programs which reside in high memory (hereafter called "RAM disks") as long as they do not interfere with TMON's exception vectors and code. If there is a RAM disk in high memory, and if it has allocated its space properly, TMON will load below it and set the top of memory address (location $10C) below itself. RAM disks, on the other hand, should respect TMON; if they don't, they will cause problems. In particular, they should not assume that $10C points to the beginning of their memory space, as it does *not* once TMON has been loaded.

If you experience problems with using TMON and a RAM disk at the same time, try reversing the order in which you load them. If you loaded TMON first, load the RAM disk first, or vice versa. If the problems persist, try to load TMON into the system heap either before or after loading the RAM disk. See the **Loading Position** section in the next chapter.

The Configuration Menus

Entering Configuration

To enter Configuration you must leave the Monitor by using Exit. You will see the *second* Main Menu. Click the Configure button, and the Main Menu will disappear and a menu bar appear:

🍎 File Edit Options

The 🍎 menu contains the desk accessories. The Edit menu is used only for the desk accessories; it is dimmed if there isn't one present. The File menu allows loading and saving of user areas to the disk, and the Options menu contains various configuration options.

The File Menu

```
🍎  [ File ]  Edit  Options
   ┌──────────────────┐
   │ Load User Area   │
   │ Save User Area   │
   ├──────────────────┤
   │ Quit         ⌘Q  │
   └──────────────────┘
```
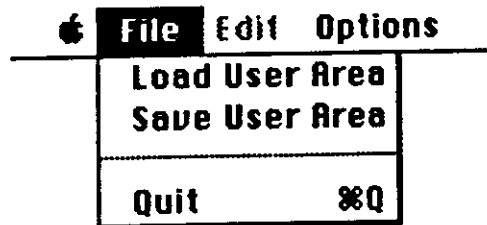
Load User Area and Save User Area load and save the current user area. If you Save a user area with the name "User Area", it will become the default user area and be loaded every time you start the Macintosh using the TMON disk or open the TMON icon from the Finder unless you use another user area's icon to start TMON from the Finder or use the Monitor... button. If there is no default user area on the disk, an internal copy of the user area with all functions present will be used instead.

☞   Loading a user area and booting TMON with a user area are not identical. When a user area is used to boot TMON, certain parameters such as screen size, user area size, and usage of A000 trap names are read from the user area and stored into TMON code. That does not take place if a user area is loaded from the Configuration mode.

The user areas are saved using the logical length given in the user area. If the logical length is greater than the physical length, the extra space is filled with zeros. If that user area is later used to boot the Monitor, its logical length will be used to set the user area's physical length.

☞   The procedure above must be followed in order to enlarge user areas.

Quit exits to the Main Menu.

Screen Buffer

> **This option controls the saving of the screen upon entry to the Monitor. You should pick "save entire screen" unless the program you are debugging won't fit in memory.**
>
> ◉ Save entire screen   (21888 bytes extra)
>
> ○ Save compressed screen   (10000 bytes extra)
>
> ○ Save compressed screen   (4000 bytes extra)   [OK]
>
> ○ Don't save screen   (no extra storage)   [Cancel]

Select the amount of memory you want the Monitor to provide for a saved copy of the screen. The more memory you give, the more of the screen you will be able to get, but also the less memory you have for the program you're debugging. On 512K Macintoshes you should almost always select the "Save entire screen" option. On 128K Macintoshes you have a tougher choice. The 4K and 10K options are the most useful, but for small graphics programs the full screen option might be necessary. The 4K option is the largest that allows you to exit to the Finder.

When you select either the 10K or 4K options, the screen is compressed. The more data you have on the screen, the less of the screen will be saved.

In some cases you might want to select the OK option even though you are not short on memory. It allows you to see clearly which areas of the screen are updated by the program you are debugging.

☞   Changing the screen size will have no effect unless you save the new setting in a user area and use that user area to boot the Monitor.

Vector Refresh

> **This option controls the refreshing of interrupt vectors. If you pick "refresh," the monitor will re-load the interrupt vectors every time it is entered. For normal use, pick "refresh".**
>
> ◉ Refresh
>   [OK]   [Cancel]
> ○ Don't refresh

Use this option only in extraordinary circumstances when the program you are debugging desires to handle some of its own exceptions. Probably the most common instances of this are the programs defining their own TRAP vectors. If you select Refresh, which is the default choice, the Monitor will keep storing its own exception vectors every time it gains control. If you select Don't refresh, the Monitor will store its vectors only once when it is initialized. The program that you are debugging may then replace the Monitor's vectors with its own vectors.

The Options Menu

```
 🍎  File   Edit   Options

             Communications
             Screen Buffer
             Uector Refresh
             A000 Trap Names
             Loading Position
             Auto-Quit
             ─────────────────
             Monitor Size   ⌘M
```

The Options menu contains several convenience settings for the Monitor. The settings are stored in the first 16 bytes of each user area, and may be loaded and saved by loading and saving user areas. The Communications and Vector Refresh options take effect immediately; the others will take effect only if you save their desired states in a user area and *boot* the Monitor with that user area. That can be done by either saving the user area with the name "User Area", or starting the Macintosh with another disk and opening that user area's icon from the Finder, or using Monitor... to select that user area to boot the Monitor. Monitor Size is not really an option because it does not allow you to change anything.

Communications

```
Baud Rate     ○ 300  ○ 1200  ○ 2400  ○ 4800  ◉ 9600

Connection    ◉ Printer Port   ○ Phone Port      [   OK   ]

Handshake     ○ XOn/XOff  ◉ DTR  ○ None          [ Cancel ]
```

Select the appropriate baud rate, port, and handshake for the printer or terminal you are using to print data in the Monitor. Click OK if you are satisfied with the choice or Cancel if you don't want to change anything. There are some communication options available which are not present here; see User Configuration Area for a more technical description of the other options.

Any changes you make are effective immediately.

It is possible to change the settings displayed here directly from the Monitor, which may be advantegous in the middle of debugging when this window cannot be invoked. See User Configuration Area for the locations of the bits in the user area that can be changed.

Auto-Quit

```
┌──────────────────────────────────────────────────────────┐
│  You may, if you wish, have TMON automatically exit to    │
│  the Finder when first started.  If you want later to     │
│  override this feature (to use Configure, for example),   │
│  hold down Option, Shift, ⌘, or the mouse button after    │
│  the screen clears during booting.                        │
│                                                           │
│  ○ Auto-quit                                              │
│                          ┌──────────┐   ┌──────────┐      │
│  ◉ Don't auto-quit       │    OK    │   │  Cancel  │      │
│                          └──────────┘   └──────────┘      │
└──────────────────────────────────────────────────────────┘
```
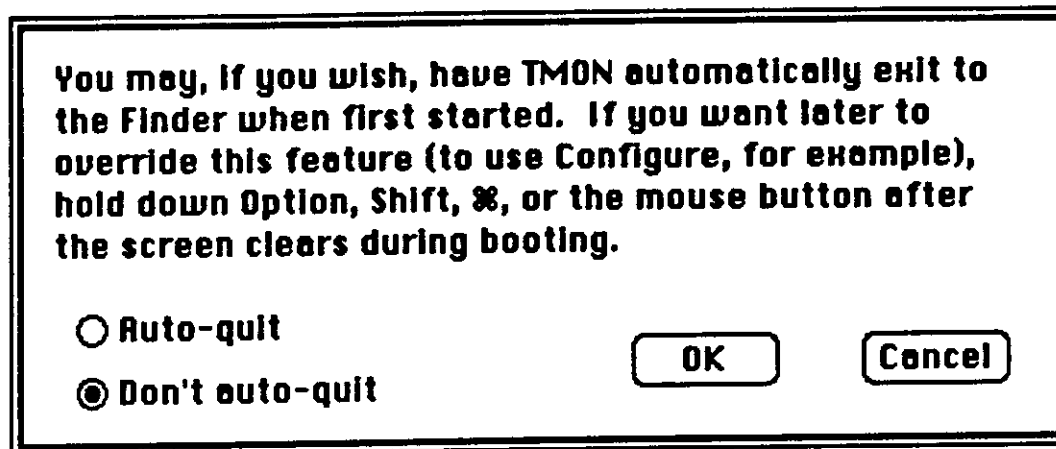
For the sake of convenience the Monitor may be loaded into your Macintosh every time you boot it using a disk that has TMON as its startup application. This is useful if you want the Monitor to be present just in case something goes wrong without having to specifically act to load it when you turn on the Macintosh. If you set this option, instead of stopping at the Monitor welcome screen, TMON displays in the middle of the screen a message stating that the Monitor has been installed and then automatically exits to the Finder. The first time the Monitor is entered it displays the reason for entering along with the welcoming message. Remember that this option is only effective if saved in a user area.

It is quite easy to override this feature if you want, for example, to use Configure. There are actually three ways: if the Finder is already running, double-click on the TMON icon to enter TMON again. If TMON is just booting, hold down either the Shift, Option, or ⌘ key or the mouse button. If you hold one down while the message "Welcome to Macintosh" disappears, you shall see the first Main Menu. If you click Monitor or Monitor... in that menu to load the Monitor, the auto-quit function will be temporarily disabled and the Monitor's welcoming message will be waiting for you.

If, on the other hand, you hold down Shift, Option, ⌘, or the mouse button after the "Welcome to Macintosh" screen disappears but before the "The Monitor is installed" message appears, the loading process shall stop when the Monitor's welcome screen appears, just as if the auto-quit function were disabled.

The auto-quit option function also works if you start TMON by double-clicking on the TMON or a user area icon in the Finder. All of the above information applies except that now all above references to the "Welcome to Macintosh" screen do not make any sense and should be replaced by references to the screen with the word "TMON" in the middle of the menu bar. It is recommended, however, that the auto-quit option only be used on a user area called "User Area" since that user area is automatically used as the default.

☞   Hold down Shift, Option, ⌘, or the mouse button while TMON is booting to temporarily override the auto-quit function.

☞   Changing this option will have no effect unless you save the new setting in a user area and use that user area to boot the Monitor.

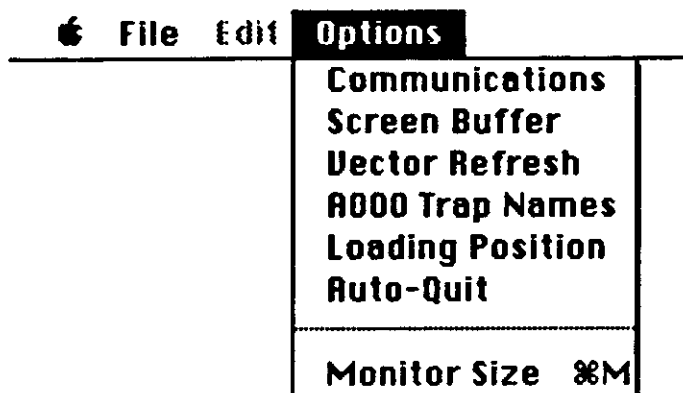Actually there are other times when the Monitor will store its exception vectors. It will do that anytime it thinks that it has been damaged and every time you enter it using the Monitor button on the Main Menu. No program except the Monitor, however, may intercept the A000 vector, trace vector, and TRAP #$F vector.

A000 Trap Names

> ## This option affects the displaying of the A000 traps by the disassembler and assembler. Using trap names is much more convenient but also takes about 3700 bytes of memory.
>
> ◉ Use trap names
>
> ○ Use trap numbers
>
> [OK]   [Cancel]

This option allows you to save about 3700 bytes of memory by not loading the A000 trap name routines and instead using the trap numbers in Assembly windows. Since there is no memory shortage on 512K Macintoshes, if you are running the Monitor on one, you should almost always use trap names.

☞ Changing this option will have no effect unless you save the new setting in a user area and use that user area to boot the Monitor.

Loading Position

> ## You have a choice of two places to load the Monitor. Loading it into high memory is the more common practice, but it prevents use of the alternate screen and sound buffers. Loading the Monitor into the system heap causes problems with a few programs.
>
> ◉ High memory
>
> ○ System heap
>
> [OK]   [Cancel]

As advertised, you may load the Monitor into either high memory or the system heap. The only reasons *not* to load the Monitor into high memory is if you are debugging a program that uses the alternate sound or video page or if it interferes with the Monitor in high memory, which should not happen. Loading the Monitor into the system heap causes the bugs in some existing applications to reveal themselves, and may be a good way to check that the program you are debugging does not have the same problems.

☞ Changing this option will have no effect unless you save the new setting in a user area and use that user area to boot the Monitor.

Predefined User Area Functions

About the Predefined User Area

The disk you received with the Monitor has a few user areas on it in addition to the predefined one inside the TMON code. These user areas are there for your convenience. They provide you with different memory and user area configurations. Examine the comments shown by the Finder's Get Info function to see the contents of each predefined user area.

This chapter will not explain the internal structure of a user area or how to create your own user functions; that is covered in the next chapter.

In the explanations below the name of the function is in boldface followed by the unabbreviated name. Any parameters used are at the end of the line.

☞ The routines described below are all present in the user area inside TMON. The other user areas on the disk contain mostly subsets of the standard user area described below in order to save memory space.

☼ Do not execute any of the functions listed below with addresses above $40FFFF, although the functions *should* reject them, avoid giving negative lengths.

☞ If you have more than one A000 trap function active at a time, the functions are executed in the following order: Trap scramble, Trap checksum, Trap intercept, Trap record, and Trap signal. If one of the functions fails and enters the Monitor, the remaining ones are not executed. Also remember that these functions are not executed on the first instruction after leaving the Monitor should that instruction happen to be an A000 trap. This is to avoid a situation where having Trap intercept set would prevent you from leaving the Monitor.

☞ The pixel in the upper left corner of the screen is turned on while one of the A000 trap intercepting functions is executing. If you press the interrupt button at that time, the interrupt will not be executed immediately but will be made pending. After the completion of the intercepting function, if an interrupt was pending, the Monitor is entered with the message "User trap: interrupt". In rare circumstances it is possible to interrupt the user area A000 trap intercepting dispatcher. If that happens, you will know it because the PC will point to the user area. Use Exit, and press interrupt again.

➪ Be very careful with using Label file load, LoadRes, and ShowScrn while one of the A000 trap intercepting routines is active. Since these routines leave the Monitor, the A000 trap intercepting functions will be executed on any traps they execute! Trap intercept, for example, will intercept the traps and go back to the Monitor with the PC and registers set to the user area. If that happens, deactivate the offending trap intercept routines and Exit the Monitor.

☼ A much worse problem than the one described in the previous note arises when one of the A000 trap intercepting routines intercepts a trap executed from an interrupt handler. One particularly common case is the interception of _PostEvent due to a mouse click or keyboard activity. *Usually* this is harmless if you realize what is happening, but sometimes the consequences can be strange indeed. Consider the following scenario, which happened to me several times: You have Trap intercept set to intercept all traps and are single-stepping through a program by typing ⌘S. Suddenly, however, you release the key at just the right time to generate a key-up event outside the Monitor (If you released the key while the Monitor was executing, nothing would happen since the trap intercepting routines are disabled then). After the _PostEvent was intercepted, the PC points to a place in ROM, and you are wondering what happened — all you did was single-step an instruction! You Exit, and only then do you arrive at the instruction following the one you stepped, but with one difference: the trace flag is now set in the status register! The Monitor uses the trace flag to single-step instructions, and it normally turns it off after the

Memory Size

The Monitor's memory usage is as follows:
 18758 bytes for the Monitor code.
  3748 bytes for the A000 trap name routines.
  1280 bytes for Monitor's variables and local stack.
  5376 bytes for the User Area (physical size).
 22080 bytes for the saved screen and cursor.

51242 bytes total.   [**OK**]

This function just prints a summary of the Monitor's memory usage. The only way to change any of the items shown above is to change the corresponding Configuration option, save the user area, reset the Macintosh, and then use the user area to boot the Monitor.

☞   Remember that this option shows the amount of memory actually used by the Monitor the way it is presently configured; it may or may not correspond with the configuration in the current user area. In other words, the other functions in the Options menu display and change the configuration in the user area, while this function displays the configuration in the code of the Monitor presently in memory, which cannot be changed without resetting the Macintosh.

This is the minimum possible memory usage:

The Monitor's memory usage is as follows:
 18758 bytes for the Monitor code.
     0 bytes for the A000 trap name routines.
  1280 bytes for Monitor's variables and local stack.
   256 bytes for the User Area (physical size).
  192 bytes for the saved screen and cursor.

20486 bytes total.   [**OK**]

If *Value* isn't found in the specified range, No Match will be displayed. If it is found, the address is given, V is set to that address, and the numbers after the colon are adjusted to allow the searching for the next occurrence of that *Value* by simply pressing Enter.

Aside from using labels, one of the most common ways of finding subroutines in your program is to search for a string that the subroutine is known to contain. In that case *Value* is usually a quoted four-letter string.

To search ROM use $400000 as the beginning address and $40FFFF as the ending address.

AlternateRegs (Alternate registers)   *Selector*

This is a set of three functions dealing with the alternate register set located in the user area.

The three functions were placed on the same line to save screen space. *Selector* is used to identify the function to be executed.

If *Selector* is zero, the Monitor's register set is copied to the user area's register set.

If *Selector* is one, the user area's register set is copied to the Monitor's register set.

If *Selector* is two, the two register sets are exchanged.

If *Selector* is none of the above, nothing happens.

The PC in the user area's register set is shown.

☞ This function is very useful in case you are anticipating a system crash in a routine you are debugging. You can save the registers and then execute the routine. Then, even if it crashes, you can restore the registers and do what you were doing previously. Also use this routine if you are testing small pieces of code from TMON. By saving registers and later restoring them you can still use Exit to exit to the Main Menu without worrying about making your routine preserve registers.

Print (Print *Dump*)   *Begin End*
Print (Print *Disassembly*)   *Begin End*
Print (Print *File*)   *File#*
Print (Print *Heap*)   *Heap#*

Print the requested information. Press the mouse button if you would like to interrupt printing in progress. This function is useful for printing memory ranges that are bigger than a single Dump, Assembly, File, or Heap window.

The *Heap#* is zero for the system heap and non-zero for the application heap.

You can cycle through the four print routines by pressing Return without typing any parameters. These four functions were placed on the same line to save screen space.

☞ The error number is stored here by both the above four routines and the Print routine in the Monitor. Look in the Print chapter for an explanation of the error numbers.

Label table at (Locate label table)   *[numberOfLabels [Location]]*

This function must be used to allocate a table for table labels. Without a table the table labels cannot be used. Whenever you invoke this function, the old label table, if present, is cleared, and, if it was previously allocated as a system heap block, it is deallocated.

If you supply no arguments, the old label table is deallocated. If it was a system heap block, the block is released. If you supply one argument, a table of size 16*numberOfLabels is allocated on the system heap. If there was enough memory for the table, its address is then displayed in *Location*. If there was not enough memory, no table is allocated, and the line is cleared.

If you supply two arguments, the table is assumed to begin at *Location*. No checks of legality of *Location* are made. *Location* must be even!

V is set to the address of the table, if one is present.

numberOfLabels should not exceed $7FF. If it exceeds $7FF, it is treated modulo $800. If it is zero, the function behaves as if no parameters were given.

instruction is complete; you don't see it being used. In this case the Monitor got somewhat confused because it was entered at an unexpected time. Afterwards it no longer remembered to clear the trace flag. No large harm has been done; you may proceed once you clear the trace flag. The moral of the story is to avoid indiscriminately intercepting traps such as _PostEvent.

BlkMove (Block Move)   *Source Destination Length*

Move a block of memory *Length* bytes long from *Source* to *Destination*. The source and destination ranges may overlap without adverse effects.

BlkCompare (Block Compare)   *Block1 Block2 Length*

Compare two blocks of memory against each other. If they match, the result is Match. If they don't match, the address in *Block1* of the mismatch is displayed and the computer looks for the first match after that memory location and puts that address after the colon along with a corresponding address from *Block2* and the number of bytes remaining. The numbers after the colon are initialized so that you can look for the next area of mismatch just by pressing Enter or Tab.

☞   The value of V is set to the address of the mismatch. You can anchor a Dump or Assembly window to V and then you won't have to type the address of the mismatch to see the area of memory around the mismatch. What's more, you can actually anchor *two* windows, one to V, and the other one to V plus an offset which is equal to the difference between *Block2* and *Block1*. This way you can look at both blocks at the positions of the mismatch.

If you have a block of memory that seems to be filled with a single hexadecimal value, you can use BlkCompare to measure how far it extends by giving it the following parameters: *Starting address Starting address*+1 7FFFF.

Fill (Fill)   *Begin End Value [ValueLength]*

Fill a block of memory with a value. The *Begin* and *End* numbers specify the boundaries of the block, inclusive, and *Value* is the number that is to be stored into the block. The fill is a byte, word, or longword fill depending on *Value*. If *Value* is less than 256, a byte fill is performed; if it is less than 65536, a word fill is done; otherwise, the fill is a longword fill. The *ValueLength* value may be used to override that by explicitly giving the length of *Value*: 1 for a byte fill, 2 for a word fill, and 4 for a longword fill.

Find (Find *byte/word* aligned)   *Value [ValueLength [Begin [End]]]*

Search for a pattern in memory. *Begin* and *End* specify the boundaries of the block that is to be searched and *Value* is the target pattern. *ValueLength* is used in the same way as in Fill, but a length of 3 is now allowed. Also, if there is no *ValueLength* and *Value* is between 65536 and 16777215, inclusive, the length will be assumed to be 3.

The default for *Begin* is 0. The default *End* is the end of RAM, which depends on the size of memory in the Macintosh. It is $20000 for 128K machines and $80000 for 512K ones. All parameters except *Value* may be omitted, in which case the entire RAM is searched for *Value*.

☞   If no parameters are supplied, the Find toggles between a byte and word aligned search. Word-aligned search is faster and is usually used to search for specific 68000 assembly language instructions, which obviously must be word-aligned. *ValueLength* is forced to be either 2 or 4 in word-aligned search.

☞   Do not use word-aligned search to search for handles or pointers because the high bytes may contain flags which cause some valid matches to be missed. Handles and pointers should be searched with a *ValueLength* of 3.

☞   The value of V is set to the address of the match. You can anchor a Dump or Assembly window to V so you won't have to type the address of the match to see the area of memory around it.

This function does not restore the screen; instead, it uses the activate/update event mechanism of the program currently executing. That mechanism must be capable of supporting redrawing of the area of the screen containing the standard file dialog.

Any mouse clicks or key presses made while this function is executing are saved in the event queue. See ShowScrn for more details.

Checksum (Checksum)   *Begin End*

Checksum generates and displays a checksum generated from that memory range. It is used to check if a memory range has been changed through time or for comparing two memory ranges. This checksum will find most transposition errors as well as substitution errors.

The *Begin, End,* and generated checksum values are also used in Trap checksum, below.

Trap checksum (Checksum on traps)   *[Trap0 [Trap1 [PC0 PC1]]]*

Whenever an A000 trap in between *Trap0* and *Trap1* located between *PC0* and *PC1*, both inclusive, is found, do a checksum on the range specified in the Checksum range. The defaults are the value of *Trap0* for *Trap1*, 0 for *PC0*, and $FFFFFF for *PC1*. A null parameter line turns off the checksumming. When an A000 trap is encountered, if the result agrees with the checksum shown in the Checksum line, the function does nothing. Otherwise, it drops into the Monitor with a user trap message stating that the checksum has failed.

Note that once the checksum fails, it is *not* automatically recomputed. If you want to see when the area of memory specified in the Checksum line changes again, you have to press Enter on the Checksum line to recalculate the checksum.

Checksum was not written with emphasis on performance. It works very fast on small ranges, but was not really designed for checksumming large ones.

See also Checksum and the note at the beginning of this chapter for general information about the A000 trap intercepting routines.

Trap record (Record traps)   *[Trap0 [Trap1 [PC0 PC1]]]*

Whenever an A000 trap in between *Trap0* and *Trap1* located between *PC0* and *PC1*, both inclusive, is found, record it in a table. The defaults are the value of *Trap0* for *Trap1*, 0 for *PC0*, and $FFFFFF for *PC1*. A null parameter line turns off the recording. The table into which the traps are to be recorded must be specified in the Record at function; otherwise, nothing happens.

The traps are recorded in a table of 16-byte entries. In each entry the first word (bytes 0,1) is the A000 trap that was recorded. The second word (bytes 2,3) contains the low 16 bits of the value of TickCount ($16A) when the trap occurred. The following longword (bytes 4-7) is the address of the A000 trap. The values of the last eight bytes vary depending on the setting of bit 11 of the trap. If the bit was 0, the trap was probably a register-based trap, and the longword values of D0 and A0 are stored in the remaining eight bytes. On the other hand, if that bit was 1, the eight bytes from the top of the stack are given.

New entries are added to the beginning of the table. All remaining entries are shifted to make room for the new entry. The last entry is forgotten. Note that unless you specifically clear the table before you exit the Monitor, the old entries will remain in it. If fewer traps than the table size were recorded while out of the Monitor, the old traps will still remain at the bottom of the table. You can distinguish them from the new traps by the time value in bytes 2 and 3.

The indicator in the curly brackets on the Record at line shows the number of new traps recorded since the last time Monitor was exited.

There are several uses for this function. One is to generally view the sequence of traps executed by the program to troubleshoot it. Another is to intercept a single trap such as _GetResource and see which resources are required by the program. Finally, this function may be used for limited performance analysis because it records the times of execution of the traps. This function does not significantly affect the running time of the program; moreover, its execution time is independent of the size of the table: recording a new trap and shifting the table is just as fast for a 2000-entry table as for a 20-entry table. Try it! The other A000 trap intercepting functions, notably Heap scramble and Checksum on large ranges, do slow the program, however.

☼ 　The system heap must be in a consistent state for this function to be used. Also, *Location*, if given, must be even and point to unused RAM.

Label add/remove   *[Label [Address [EndAddress]]*

Add and remove labels in the label table. The label table must be allocated. If no parameters are given, do nothing except clear the result information. The result information is displayed between the two curly brackets and is either blank if nothing was done or gives information about the last operation completed. *Label* must be a label enclosed in double quotes (not an expression!). It is the label upon which the operation is being done.

If only *Label* is given, it is removed from the table if it was present; otherwise, nothing happens. The appropriate information is displayed between the curly brackets.

If *Label* and *Address* are given, *Label* is added (or replaced if it already exists) to the table. *Address* is the address assigned to it. If the *Scan resources* option in the Options window is set and *Address* falls inside a resource, the label is stored as a resource-relative label in the table, and needs no explicit ending address. A message stating that the label has been added relative a given resource type is shown between the curly brackets.

If all three parameters are provided, or the label could not be stored as resource-relative, it is stored as absolute starting at *Address* and ending the recognition range at *EndAddress*. If *EndAddress* was not provided, it is set to *Address*+$800. *EndAddress* is the first byte *past* the recognition range, not the last byte of it. A message stating that the label has been added as an absolute is shown between the curly brackets.

If the table is already full, the label is not added, and nothing appears between the curly brackets.

☞ 　When adding labels this routine automatically determines whether a label should be stored resource-relative or absolute. If you think you need more control, modify the label table directly. Add some dummy labels to it and then use a Dump window to change their data.

Label file load (Load .Map label file)

This function reads a .Map file and extracts from it labels that are inserted into the label table. Labels must be enabled and a label table must be allocated. When executed, this function leaves the Monitor and shows a standard file selection dialog. All files of type TEXT are displayed. If you press Cancel, the Monitor is reentered with the message "Bad load". If you select a file, it is opened and read. Any errors cause a return to the Monitor with the "Bad load" message. If the label table becomes full, any extra labels in the file are ignored.

The function has definitely not been optimized for speed in order to keep it simple. It should not be difficult to change it to read other file formats.

This is the format of the TEXT file. All spaces are competely ignored; they are not used as delimiters of any kind and are removed from wherever they appear. This means that they can appear in the middle of numbers or labels without being detected. Any string of characters beginning with a character below $20 or above $7E and ending with an equal sign (=) is considered a label. The two delimiters are, of course, not included in the text of the label. The label, in order to be entered into the label table, must be followed by two hexadecimal numbers separated by another delimiter, usually a colon. The first number is 0 for absolute labels and gives the CODE segment number for CODE resource-relative labels. The second number gives the value for absolute labels and gives the offset inside the CODE resource minus 4 for the relative labels. No range checking is performed. The recognition range ending address for absolute labels is set to the label address plus $800. If the label already exists in the label table, the old one is replaced. Resource-relative labels relative to resources other than CODE cannot be specified in .Map files.

☼ 　This function is by far the least reliable one in the entire Monitor. It requires most of the managers like QuickDraw, the Window Manager, Dialog Manager, Font Manager, TextEdit, and others to be initialized. Moreover, it uses the standard file dialog to select the file, which is very risky to do from a Monitor. For all these reasons it is recommended that you use this function only from *TMON when the monitor first appears*. The only way to *assure* a consistent state is to execute this function the first time the Monitor screen appears (with the welcoming message) or to click Monitor in the Main Menu.

This function learns that Option-interrupt was pressed because the Monitor tells it that. If you press Option-interrupt while this function is disabled or the Monitor is active, nothing happens.

☞   Press Option-interrupt to activate the smart trap intercept. It is also recommended that you get the habit of using the right Option key. If you use the left one, you risk accidentally pressing the ! key too, which would have disastrous consequences (It does a complete Monitor reset).

See also the note at the beginning of this chapter for general information about the A000 trap intercepting routines.

Trap scramble (Check, Scramble, and/or purge the Heap) *[Zone#]*

Scramble the heap on A000 traps. Unlike the other A000 trap intercepting routines, this one does not give you a choice of traps on which it is executed. The heap is scrambled whenever *a trap that might trigger a heap compaction* in this range is intercepted; the Monitor is not entered. The traps that might trigger a heap compaction are: _NewPtr ($A01E), _NewHandle ($A022), _ReallocHandle ($A027), and _SetPtrSize ($A020) and _SetHandleSize ($A024) if the new length is greater than the old length. The Monitor is entered if the heap is somehow damaged (If that happens, the heap will have been partially scrambled up to the location of the error.)

Zone# selects the heap to be scrambled. Use 0 for the system heap and any non-zero number for the application heap. The scramble is enabled when the zone number is visible. Press Return on the line once to turn off the scrambling.

The "heap scramble" in the preceding two paragraphs may not really be a heap scramble. There are four choices possible; they are chosen by consecutively pressing Return on the line.

The *Check* choice just checks the heap for consistency without modifying it. It is useful for locating routines in your program that somehow damage the heap.

The *Check, purge* choice first purges all purgeable blocks from the heap and then checks it.

The *Check, scramble* choice checks and moves as many relocatable blocks around the heap as possible. This is used for finding handle dereferencing errors in programs, which are surprisingly common. The check and scramble are done simultaneously; a part of the heap may have already been scrambled when an error is found; nevertheless, the heap area immediately before and after the error is never scrambled. This option is very highly optimized for speed within the constrains of space in the user area; still, it is quite slow.

Note that this option also clears all free blocks in the heap except in some cases the last one (to make the speed bearable; the last one is usually very large), but not to zeros. This provides additional assurance that dereferencing errors are eliminated.

The *Check, scramble, purge* choice does all three: a purge, a check, and a scramble.

A heap scramble consists of moving as many unlocked relocatable blocks in the heap as possible. This way a heap compaction is simulated every time one *could* happen. Any handle dereferencing errors which otherwise would be rare and random and very difficult to find because they would occur only on heap compactions are now made to occur consistently every time, making them much easier to find.

In addition to moving the free blocks in the heap the heap scramble erases (to a nonzero value) all free blocks and consolidates any consecutive ones. The only free block that may not be completely erased is the last one. This is done for performance reasons.

It is possible that some relocatable blocks will not be moved. This will occur in the rare situation that one relocatable block is caught between two immovable ones.

Scramble now (Scramble Heap Now)

Check and possibly scramble or purge the heap as described above according to the options currently set in Trap heap scramble. The action is performed immediately. The last heap zone entered into Trap heap scramble is used; if Trap heap scramble was never used, the system zone is used.

☞ You will probably want to use a Dump window to view the record and a Number window to find the trap name from the number given in the records Be aware, however, of one danger: do not type values such as $A122 (_NewHandle) into the Number window and expect to see the right trap name on the bottom line. Since the A000 trap name routine uses the low 9 bits of the trap number, it thinks that you want trap $122 ($A922), which is _BeginUpdate. This problem occurs only with eleven A000 traps which have bit 8 set as an option. It is addressed further in the sections about Numbers and Number and Assembly windows.

See also Record at and the note at the beginning of this chapter for general information about the A000 trap intercepting routines.

Record at (Where to record traps) *[fullStop nMessages [Location]]*

This function must be used to allocate a table for Trap record. Whenever you invoke this function, the old label table, if present, is cleared, and, if it was previously allocated as a system heap block, it is deallocated.

If you supply no arguments, the old label table is deallocated. If it was a system heap block, the block is released. If you do not supply *Location*, a table of size 16*nMessages is allocated on the system heap. If there was enough memory for the table, its address is then displayed in *Location*. If there was not enough memory, no table is allocated, and the line is cleared.

V is set to the address of the table, if one is present.

If you supply all arguments, the table is assumed to begin at *Location*. No checks of legality of *Location* are made. *Location* must be even!

nMessages should not exceed $7FF. If it exceeds $7FF, it is treated modulo $800. If it is zero, the function behaves as if no parameters were given.

fullStop is a flag for use by Trap record. If it is zero, recording takes place until the Monitor is entered again. If it is nonzero, the Monitor is automatically invoked via a user trap at the moment the table overflows. The trap that would cause the table to overflow is not recorded.

If you have already allocated the table using a *Location*, you may clear it by pressing Enter on the line. You *do not* want to press Enter on the line if it was allocated as a system heap block, because the routine will believe that you are now giving it an address! To clear a table that is a system heap block, press Return with the cursor between *nMessages* and *Location*.

The indicator in the curly brackets shows the number of new traps recorded since the last time Monitor was exited.

※ The system heap must be in a consistent state for this function to be used. Also, *Location*, if given, must be even and point to unused RAM.

Trap intercept (Intercept Traps) *[Trap0 [Trap1 [PC0 PC1]]]*

Intercept all A000 traps with the trap number between *Trap0* and *Trap1*, inclusive, and which are in the block of memory from *PC0* to *PC1*. The defaults are the value of *Trap0* for *Trap1*, 0 for *PC0*, and $FFFFFF for *PC1*. A null parameter line turns off the checksumming. When a trap is encountered, the Monitor is entered with a user trap message. See also the note at the beginning of this chapter for general information about the A000 trap intercepting routines.

Trap signal (Smart interrupt) *[Trap0 [Trap1 [PC0 PC1]]]*

Like the other A000 trap intercepting functions, this one executes on A000 traps with the trap number between *Trap0* and *Trap1*, inclusive, and which are in the block of memory from *PC0* to *PC1*. The defaults are the value of *Trap0* for *Trap1*, 0 for *PC0*, and $FFFFFF for *PC1*. A null parameter line turns off this function.

Unlike the other trap intercepting functions, this one does nothing most of the time. It remains dormant until you press interrupt while holding down the Option key. Once you do that, this function will drop into the Monitor as soon as it is executed. It is very useful for stopping your program at a specific point as opposed to anywhere. Some possibilities are setting *Trap0* to _SystemTask or _GetNextEvent or setting *Trap0* and *Trap1* to all traps but restricting the PC to your main program.

Creating Your Own User Functions

Technical User Area Description

The user area is a variable-length block of memory reserved by the Monitor. It has two purposes: to allow you to use the predefined and add your own functions to the Monitor and to store the Configuration setting. The Configuration setting, a few other parameters, and the user area identification number are all stored in the first 48 bytes of the user area. They are followed by a linked list of names and other parameters for the routines. The routines themselves are at the end of the user area.

The user area routines must be relocatable. The user area is placed between Monitor's variables and the Monitor's stack, which could be anywhere in memory. Once the user area is loaded, however, it is never moved (but it could be saved and loaded into a different place).

This chapter is organized roughly in the order of the data in the user area. The description of the configuration bytes is first, followed by the descriptions of some special numbers and vectors that are also stored in the Configuration area. Then the names and finally the routines are described.

It is difficult to learn how to create your own user routines without looking at examples. You are encouraged to look at default user area source file (supplied on the disk) for any ideas.

All numbers are in decimal unless preceded by a dollar sign or the context implies that the number is hexadecimal. Bit 7 is the most significant bit of a byte and 0 the least significant.

⇨ Be careful with this function (Scramble Now). If the program you are debugging has any dereferenced handles at the time you invoke it, that program may later fail. If you are not sure whether this is the case, use Intercept Traps to find the nearest A000 trap that might cause a heap scramble (see previous section). Before such a trap it is definitely permissible to scramble the heap.

MemHeap (Free Memory Left on Heap)   *Zone#*

Display the total amount of free bytes on the heap, the maximum number of contiguous bytes, and the number of bytes the heap may grow. The system heap is used if *Zone#* is zero, otherwise the application heap is used.

☞ The heap zone is compacted and all purgeable blocks are purged from it.

⚕ Do not use this function if the heap zone is inconsistent (has invalid blocks in it). Also avoid using it if the application program you are debugging does not expect the heap zone to change.

LoadRes (Load Resource)   *Type ID*

Load a resource into memory. Search for the given *Type* and *ID* in all open resource files in the order defined by the Resource Manager. If the resource doesn't exist, do nothing. If the resource is already in memory, give its address. Otherwise attempt to read the resource from the disk. If there is no error, load the resource and give its address.

This routine leaves the Monitor for technical reasons (to prevent problems with crashes and disk swapping). It reenters the Monitor as soon as the resource is read.

☞ If the resource is in a file on a disk not currently in any drive, you may be asked to insert the disk.

⇨ Do not press the interrupt button while this routine is executing. Also, any mouse clicks or key presses made after the Monitor has been left are saved in the event queue. See ShowScrn for details.

⚕ The heap must be in a consistent state for this routine. This routine may cause heap compaction.

ShowScrn (Show Screen)

ShowScrn leaves the Monitor and stays in a routine that does nothing except check for a mouse button click and reenter the Monitor when the mouse button is pressed. This function also allows you to move the mouse cursor in the program you're debugging without having the program react to that motion. That could be useful sometimes.

This function will fail with a system error $1C if the stack pointer is below the top of the application heap.

⇨ Do not press the interrupt button while this routine is executing.

☞ Any mouse clicks or keystrokes are recorded as events, but since the Monitor does not use the event queue, they will remain queued until you exit the Monitor. This includes the mouse click which you use to return to the Monitor. Although it may be annoying at times, this side effect can also be very useful for testing how the program you are debugging responds to a rapid succession of events. You could, for example, use ShowScrn to click on several buttons of a dialog (you would have to use ShowScrn several times for this), and then exit the Monitor and see how the program responds. This method can discover some very subtle errors.

Reset (Reset the Macintosh)

Eject all mounted disks and reset the Macintosh. This is better than pressing the reset button because it ejects and unmounts all disks. Although it should work every time, this function is not guaranteed on hard disk drives — test it. In order for Reset to succeed, the volume queue and file variables must be in a usable state.

Names and Local Storage in the User Area

At the 14th byte of the user area is a pointer to a linked list of user routine descriptors. The list is composed of entries shown below:

First is the offset from the beginning of the user area to the next routine's descriptor in the list. The offset is a word. Next is another word offset from the beginning of the user area, this time to the starting address of a user routine. It is followed by the length of the routine's name and then by the name itself (described in the next section). An extra byte may be inserted after the last character of the name to make the next field on a word boundary. The parameter count byte is after the name. It contains a bit map indicating the number of parameters accepted by this routine. After that is a byte containing the length of the user routine's local variable space followed by the variable space itself. (The length of the local variable space is no longer used. It was once used by earlier versions of TMON, but the current one ignores it. It may be used again in the future.) The data after the variable space can consist of anything; in particular, it may be the routine code, or the name record of the next routine.

What's in a Name?

The names are usually not pure ASCII strings, although they may be. Usually they are much more complicated, including displaying of variables, ASCII values, or even conditionals. This is a powerful feature of the Monitor that simplifies creating your own user areas. Some functions described below refer to a pointer called P. It is an internal Monitor variable that is initialized to the beginning of the user routine's local variable space every time before the name is printed. In the following section "printing" means displaying on the screen, not the printer. Some excellent examples of control sequences in names are present in the source code of the default user area. You are encouraged to look at these to better understand "names" as described below. Here is a table of the "control codes" that can be used in a name:

$00	EndIf
	Cancel a preceding IfElse, IfPos, or IfNeg.
$01	IfElse
	Flip the condition of the last IfPos or IfNeg.
$02	IfPos
	If the byte at P is positive, execute the next section of code. If it is negative, skip until the next EndIf or IfElse. In either case increment P. The conditionals may be nestable to any reasonable level. Each IfPos may be optionally followed by IfElse, but must be followed by EndIf with one exception: it is not necessary to put an EndIf before the end of the string.
$03	IfNeg
	If the byte at P is negative, execute the next section of code. If it is positive, skip until the next EndIf or IfElse. In either case increment P. The conditionals may be nestable to any reasonable level. Each IfNeg may be optionally followed by IfElse, but must be followed by EndIf with one exception: it is not necessary to put an EndIf before the end of the string.
$04	Colon
	This is one of the more powerful commands in names. It prints a colon, but has a much more profound side effect. Everything printed after this command will appear to the right of the colon and provide a default for the user's editing. Some pre-defined user functions that use this control code are BlkCompare, Find, Checksum, LoadRes, and Intercept traps.
	No more than one Colon can be interpreted, although more than one may be present (using conditionals). Any Colon(s) encountered after the first one is interpreted, are ignored. If there is no Colon in the name, an implied one is inserted after the end of the name, so that there always is a colon on the line.
$05 to $0E	Skip
	Increment P by 1 to 10 bytes. $05 increments it by 1 byte, $06 by two bytes, etc.
$0F to $16	PrHex
	Print from one to eight hex nibbles from the memory pointed by P. P is incremented past the byte that contains the last nibble printed. For example, $13 causes the least significant nibble of the byte at P and both nibbles of the two following bytes to be printed as a five-digit hex number.

The User Configuration Area

As stated earlier, the first 48 bytes of a user area are used to hold the current Configuration setting and some other interesting data. Here is a summary of this data; some of the items will be explained in more detail later.

BYTE	BIT RANGE	DESCRIPTION
0-1	all	The length of the user area. It must be a multiple of 256, less than $6000, and can not be zero.
2-3	all	This user area's version/ID number. This number is for identification purposes only; no part of the program references it.
4	7	0 if the vector refresh is on; 1 if it is off.
4	6	0 if A000 trap names are to be loaded upon booting; 1 if not.
4	5	0 if the Monitor is to be loaded into high memory upon booting; 1 if system heap.
4	4	1 if auto-quit is enabled; 0 if not.
4	3	Reserved. Must be zero.
4	2-0	The amount of screen compression used; 0 indicates saving the entire screen (21888 bytes), 1 a 10K compression, 2 a 4K compression, and 3 no compression. The values from 4 to 7 are illegal at the present time. This information is used only during booting.
5	7	Set to 1 to inhibit calling the user identification routine for heap windows.
5	6	Set to 1 to inhibit scanning resources for heap windows.
5	5	Set to 1 to inhibit table labels.
5	4	Set to 1 to inhibit embedded name labels.
5	3	Set to 1 to inhibit scanning resources for label routines.
5	2	Set to 1 to inhibit pseudo-label identification.
5	1	Set to 1 to inhibit all labels (master switch).
5	0	Reserved. Must be zero.
6	7	The port used for printing. 0 is the printer port; 1 is the phone port.
6	6-0	Reserved. Must be zero.
7	7-2	Reserved. Must be zero.
7	1-0	The handshake used for printing. 0 is none, 1 is XON/XOFF, and 2 is DTR. A value of 3 will cause unpredictable results.
8	7-6	The number of stop bits used by the serial printing routines. 1, 2, and 3 are 1, 1.5, and 2 stop bits, respectively.
8	5-4	0 and 2 are no parity; 1 is odd, and 3 is even parity for printing.
8	3-2	0, 1, 2, and 3 are respectively 5, 6, 7, and 8 bits per byte for printing.
8-9	1-0;7-0	A constant determining the baud rate being used. The value is 115200 / baud rate - 2. For instance, use 4 for 19200 baud, 10 for 9600 baud, and 46 for 2400 baud. The constant is ten bits long.
10-11	all	Offset in the user area to an A000 hook routine; 0 if there is no such routine. For example, if the user area starts at $7000, and the A000 handler routine within the user area starts at $732D, the value stored here would be a $032D.
12-13	all	Offset in the user area to the location to store the Print error code or 0 if there is no such location. Whenever the Print function is used from the menu or the predefined
	user	area, the error code is stored at this word in the user area.
14-15	all	Offset to the first user routine's description block or 0 if there is none.
16-17	all	Offset to the heap window identification routine or 0 if there is none.
18-19	all	Offset to the user area initialization routine or 0 if there is none.
20-21	all	Offset to the user area entry routine or 0 if there is none.
22-23	all	Offset to the user area exit routine or 0 if there is none.
24-25	all	Offset to the user area label table recognize routine or 0 if there is none.
26-27	all	Offset to the user area embedded label recognize or 0 if there is none.
28-29	all	Offset to the user area label table evaluate routine or 0 if there is none.
30-31	all	Offset to the user area embedded label evaluate or 0 if there is none.
32-47	all	Unused; reserved for future expansion. Must be zero.

All other registers contain zeros. The user routine does not have to preserve any registers except, of course, A7. The interrupt level is set to zero upon entry to the user routine, but the user routine may set it to anything it wishes. The status register does not have to be preserved either.

There are, however, certain restrictions on the user routines that are called from the Monitor. They must not cause any exceptions; in particular, this includes address and illegal instruction errors and trace interrupts. There are some ways user routines may get around these restrictions if it is absolutely necessary, as described in the next sections.

The A000 Trap Intercepting Hook

At the beginning of the user area there is a word that contains either zero or an offset to a user A000 trap intercepting routine. This makes functions like Intercept traps and Scramble heap possible. The user routine pointed by the vector is executed before every A000 trap *except* traps that occur while the Monitor is executing (See the Self-Check section for a definition of when the Monitor is executing) and traps that occur while the first instruction is executing after leaving the Monitor (See **Trace Flag Side Effects** for more information on this topic).

The A000 trap routine must preserve all registers except the CCR. Upon entry all registers are the same as before the A000 trap except A5, which has been saved on the stack. A5 is initialized to the beginning of Monitor's variables. The standard user area contains a routine which is usually linked to this hook; it is called INTERCEPT. It obtains the trap number and PC and dispatches any user A000 intercept routines that are active.

The trap routine must either return to the routine that called it using RTS or restore all registers to their original states (this includes the A5 that was saved on the stack) and execute a TRAP #$F instruction, as described in User Routines Entering the Monitor. Examine the INTERCEPT routine for a safe way of doing that.

User Routines Leaving the Monitor

A user routine may leave the Monitor if it wishes to do so. It must initialize the register area in the Monitor's variables to the values the registers are to assume after the Monitor has been exited. This includes in particular the program counter, stack pointer, and status registers. After doing this the routine must execute a JMP -12 (A5) instruction. A5 must contain the address of the Monitor's variables. At that time the Monitor executes a kind of an automatic Exit function. The register area in Monitor's variables will be described in a later section. Refer to the ShowScrn and LoadRes routines in the user area source for examples of leaving the Monitor and re-entering it later, although that is not the clearest example because before leaving the Monitor the return status register and program counter values are already pushed onto the stack in anticipation of the TRAP #$F instruction that will be executed after the mouse button is pressed.

User Routines Entering the Monitor

User routines that are called from outside the Monitor or have exited the Monitor may reenter it by pushing a longword and a word on the stack and executing TRAP #$F *inside the user area*. That TRAP #$F instruction is interpreted as an attempt to reenter the Monitor only if it is located within the user area.

The longword pushed onto the stack is the value that will be loaded into the program counter in the Registers window. The word pushed onto the stack after the longword is the status register value that will be placed into the same window. The other registers in the Registers window come directly from the values left in the registers. Some examples of the usage of this feature are the ShowScrn routine and the routines that exit to the Monitor after a trap has been intercepted or an error in the heap found.

The Heap Window Identification Routine

Locations $10 and $11 of the user area contain a word offset from the beginning of the heap to the Heap window identification routine that is described in the Heap window section. The routine is used to identify heap blocks and may be customized. The routine is given a flag indicating the type of block in D3. A zero is a non-relocatable block, one is a non-resource relocatable block, and two is a relocatable block that has already been

$17 to $1E   **PrASCII**

Print 1 to 8 ASCII characters from memory starting at P. P is incremented past that memory block. ASCII values lower than $20 or higher than $7E are printed as tiny periods.

$1F   **NoOp**

No operation.

$20 to $7E   Print that ASCII character.

$7F   Print a tiny period.

$80   **DisAsm0**

Print the name of the A000 trap whose number is in the word at P. P is incremented past the word. The number at P is decoded in the same manner as in a Number window: bits 0 to 8 contain the trap number, and bits 9 to 15 are ignored. If the trap names are not present or the trap given has no name, its number (plus $A000 or $A800, depending on the trap) is displayed instead, preceded by a dollar sign.

$81   **DisAsm1**

This is the same as DisAsm0 except that the word at P is decoded in the manner an Assembly window would show it as opposed to a Number window. All 16 bits are significant; bits 12 to 15 should contain $A. A space and the hexadecimal digit indicating the value of bits 8 to 11 are printed when that value differs from the default. If the trap names are not present or the trap given has no name, this function is equivalent to the sequence $ PrHex+4.

$82   **Recognize**

Call the label recognize routine on the address given in the longword at P, and report the results. Up to 23 bytes of the destination string might be used. Nothing is displayed if the address could not be recognized.

$83 to $FF   Unimplemented. Currently behave as NoOps.

The line is truncated to 84 characters including the colon. Names that are too long may provide too little editing space for the user.

Parameter Count

The parameter count byte defines the number of parameters allowed by the routine. A parameter is a number or an expression typed by the user on that routine's line. The parameters are separated by spaces and are automatically evaluated by the Monitor. The parameter count byte is actually a five-bit bit map contained in the least significant bits of the byte. The three most significant bits are ignored at the present time. If bit 0 of the byte is set, the user routine may be called with no parameters; if bit 1 is set, it may be called with one parameter, etc. If the routine allows more than one amount of parameters to be present, it may find the number of parameters actually given and provide defaults for the missing parameters.

Bit 7 of the parameter count serves a special function. If it is set, a label is expected instead of the first two parameters. The label is checked for syntax but not evaluated. The first four characters of the label are given as parameter 0 (in D0), and the second four are given as parameter 1 (in D1). If there are less than eight characters, the missing ones are padded with blanks; if there are more, the extra ones are ignored. The label counts as two normal parameters.

Register Conventions

Upon entry to the routine the following values are present in the 68000 registers. All values are longwords.

D0 to D3	Parameters provided by the user or zeros if not present.
D7	The number of parameters supplied by the user (0 through 4).
A0	A pointer to this user routine's local variable storage.
A1	This subroutine's starting address.
A2	The user area's starting address.
A5	A pointer to the Monitor's variables (described later).
A7	Monitor's stack pointer. At least 200 bytes are available on the stack.

The User Enter and Exit Routines

These user routines are called after entering and before exiting the Monitor. Their offsets are in bytes $14 and $15 for the enter and $16 and $17 for the exit routine. The routines do not have to preserve any registers except the high byte of SR and A7. On entry A5 points to the beginning of the Monitor's variables, and A7 points to the Monitor's stack. Look in the standard user area for examples of usage of these routines.

Some very large problems could arise if the user enter routine causes an address error. The Monitor will re-initialize itself, and, in the process, will call the enter routine again, causing another error. The cycle will thus continue. The only way to break out of the cycle is to press Option-⌘-interrupt, but even that may not always work (especially if the keyboard handler is no longer operational, in which case it will not detect the Option and ⌘ key being pressed).

The User Label Routines

There are four user routines which are used by the Monitor's label system for label evaluation and recognition. Two of these routines do recognition: _LSCAN, at $18 and $19, for label table recognition; and _CSCAN, at $1A and $1B, for embedded name recognition. _LFIND, at $1C and $1D, evaluates table labels; and _CFIND, at $1E and $1F, evaluates embedded name labels. These routines are called only if labels are enabled and if they are not inhibited by Options. The register conventions are listed in the user area source code. These routines should take measures to avoid crashing on address errors. They should also be designed efficiently, as slow ones will excessively slow the Monitor.

The Monitor's Variables

This is an incomplete list of the Monitor's variables. Only the more useful variables are shown. You can learn how to use them by examining the user area source file. The locations of variables are offsets from A5. The lengths of the variables follow the locations.

$15 (B)   MONEXECUTING
               This byte is used to decide whether the Monitor is currently executing or not. $6B means it is
               executing, $29 means that the first instruction after an exit of the Monitor is executing, and any
               other value means that the Monitor isn't executing. Do not change this value unless you are sure
               what you are doing; the A000 trap intercept routine in the user area source contains an example of
               changing MONEXECUTING.
$1A (L)   EVENTINTERCEPT
               Saved address of the OS Event Manager routine while the Monitor is executing.
$1E (L)   DESNIFF
               Saved value of $110 while the Monitor is executing.
$22 (B)   NWINDOWS
               The number of Monitor windows currently on the screen (includes alert windows).
$23 (B)   BREAKPTMAP
               A bitmap of the breakpoints. The MSB is used internally by the Monitor to indicate if the
               breakpoints are set at the present time or not. The seven least significant bits are set if the
               corresponding breakpoints have been set.
$24 (L*7)   BREAKPOINTS
               The addresses of the breakpoints or zeros if the breakpoints are reset.
$40 (W*7)   BREAKSAVES
               The values of the words "under" the breakpoints.
$4E (L)   REG.PC
$52 (W)   REG.SR

identified as a resource. Free and invalid blocks are not passed to the identification routine. The user identification routine will probably want to ignore type-2 blocks.

D3 contains the flag described above.

A0 contains the address of the beginning of the user area.

A1 contains the address of the heap zone to which the block belongs.

A2 contains a pointer to the area in which the text identifying the block is to be stored. Look at the listing of the default user identification routine to learn how to handle A2. Make sure that you do not run off the right edge of the string at A2.

A3 is a pointer to the heap block to be identified.

A4 is another pointer to the destination area. The difference between it and A2 is that while A2 points to the free space after the last word already present in the string, A4 always points to the same place: the position after the second space after the size correction digit.

A5 points to Monitor's variables.

A6 contains the address of the handle for relocatable blocks only.

A7 is the stack pointer. At least 60 bytes are free on the stack.

The routine may destroy the contents of any registers except A2, A5, and, of course, A7. If the routine was able to identify the block, it should move A2 past the information it has written into the destination.

⇨ The routine should not take much more than 1/100th of a second to execute; longer times will tend to excessively slow the Monitor.

※ Make sure that the routine does not write outside the designated destination area. See the listing of the supplied routine for details.

※ The routine should check all data structures it uses to avoid address errors and following NIL pointers. It should not assume that anything is correct except in extremely time-critical cases.

The User Initialization Routine

There exists a routine in the user area that is executed immediately *before* the Monitor is first initialized and immediately before the Monitor is reinitialized after clicking the Monitor button in the Main Menu if the Monitor is already present. Locations $12 and $13 of the user area contain a word offset from the beginning of the user area to the beginning of this initialization routine. These two bytes contain zeros if there is no initialization routine.

This routine is called *before* the Monitor initializes itself; this means that none of the Monitor's variables contain valid information. This also means that the Monitor's self-checking has not yet begun and, therefore, the Monitor can be patched without generating the message stating that the Monitor has been damaged. In fact, the main purpose for including this routine is to allow user areas that modify the Monitor to be made. A secondary purpose is to allow self-initializing user areas.

The routine does not have to preserve any registers except the high byte of SR and A7. On entry D0 contains 0 if the routine is called the first time and -1 everytime thereafter (It can be called more than once if the Monitor button is used to re-initialize the Monitor). A5 points to the beginning of the Monitor's variables. None of the variables themselves, however, have been initialized. A7 points to the application program's stack, not the Monitor's stack.

☞ This routine is *not* called if ⌘-interrupt is pressed or the Monitor reinitializes itself due to one of the conditions listed in the Self-Check section of the Exception Handling chapter.

※ Make sure you know exactly what you are doing before attempting to patch the Monitor! Remember that none of Monitor's variables contain valid information when the initialization routine is executed. Your initialization routine also should not call any of the Monitor's routines if you are not sure whether such routines depend on the initialization of Monitor's variables.

-$18(A5)   _PUTASCII

Given a byte in D0 and A2 pointing to a destination area, _PUTASCII stores either D0 if it is a valid ASCII character or $7F (a tiny period) if it isn't in (A2). A2 is incremented one byte. D0 is destroyed.

-$1C(A5)   _PUT1DIG

-$20(A5)   _PUT2DIG

-$24(A5)   _PUT4DIG

-$28(A5)   _PUT6DIG

-$2C(A5)   _PUT8DIG

These routines take either the hex digit, byte, word, three bytes, or long word in D0 and put it in hexadecimal format at (A2). A2 is incremented by the corresponding number of digits. D0 and D1 are destroyed.

-$30(A5)   _NEXTCRESFILE

Find the next file in the linked list of resource files. Check for NIL handles and address errors. Ignore files with address errors caused by accessing type maps. On entry D1 contains a handle to the next file. On exit, if Z if set, there is no next resource map. If Z is clear, A1 points to the type list, D0.W contains the file reference number, and D1 has a handle to the next file. No other registers are affected.

-$34(A5)   _FINDRES

Find a resource given its type and ID. Check for address errors and NIL handles. On entry D2 contains the type and D3.W the ID. On exit, if D0 is -1, the resource was not found (maybe not loaded). If D0 is 0, D2 points to the resource, and A0 has the handle to the resource. D1, D4, A0, and A1 are destroyed.

-$38(A5)   _RECOGNIZE

Call the recognize routine. On entry D2 contains the address to be recognized, and A2 must points to a destination string at least 23 bytes long. Upon exit, A2 is advanced past the recognition data stored in the string, and D0 is either 0 if the recognize was successful, or -1 otherwise. D1-D4, A0, and A1 are destroyed. All of the Options switches are obeyed.

$54 (L)   REG.USP
$58 (L)   REG.D0
               Registers D1 through D7 follow.
$78 (L)   REG.A0
               Registers A1 through A6 follow.
$94 (L)   REG.A7
$98 (L)   REG.NUM
               The current value of N.
$9C (L)   REG.V
               The current value of V. Many user area routines change this location.
$A0 (10*30)   WINDOWS
               The main array of windows. Up to ten windows may appear simultaneously on the screen.
               The first window is the topmost. Each window record is 30 bytes long, and its first byte indicates
               the type of a window: $00 Alert, $01 Registers, $02 Breakpoints, $03 User, $04
               Number, $85 Dump, $86 Assembly, $87 File, $88 Heap, and $09 Options. Type $0A is
               used internally. The other bytes of a window record indicate the window's length and position on
               the screen, and in some cases the data displayed in the window. Assembly window records, for
               instance, contain the lengths of the instructions displayed in them.
$1F7 (B)   MONTRACETIME
               Information on what to do if a trace interrupt occurs and MONEXECUTING is $29. 0 means
               enter the Monitor with trace flag clear, 1 enter the Monitor with trace set, $80 put in the
               breakpoints and leave, and $81 is used by GoSub to step through the JSR or BSR instruction.
$1F8 (L)   SYSERRVECTOR
               The saved value of the system error vector. Jump to the address stored here to generate a
               system error.
$200 (L)   SYSA000VECTOR
               The saved value of the system A000 line exception vector.
$206 (B)   USEDPAGE0
               0 if the standard screen page is to be used after exiting the Monitor, $FF if the alternate page
               should be used.
$207 (B)   USERIINFORM
               This variable is cleared every time the Monitor exits. It is set to $FF if the interrupt button is
               pressed (without Option or ⌘ keys) while MONEXECUTING is $6B. USERIINFORM is also set
               to $01 whenever Option-interrupt is pressed (without the ⌘ key). This variable is examined by
               the user area INTERCEPT routine to determine if the interrupt button was pressed while one of its
               dispatched routines was executing. It is also used by the Trap Signal function.
$208 (8*B)   ALCPCORDER
               The order of allocation of ALCPCVALUES. 0 is the next to be allocated.
$210 (8*L)   ALCPCVALUES
               Up to eight return addresses for up to eight recursive invocations of GoSub or Step. See the
               section on GoSub and Step for more details.
$500   USER
               This is the beginning of the user area.

The Monitor's Vectors

This a list of Monitor's vectors which may be accessed by jumping to or calling subroutines at *offset(A5)*.

-$08(A5)   _PRINT1
               This is for use of the printing routine only. See PRINT in the listing of the default user area.
-$0C(A5)   _EXITMON
               This routine is used for exiting TMON and has been described earlier.
-$14(A5)   _PRINT2
               This is for use of the printing routine only. See PRINT in the listing of the default user area.

Macintosh Memory Management

Introduction

It is assumed that the reader has a 68000 assembly language manual, and no explanations of 68000 assembly programming will be made.

This chapter contains an overview of the memory and resource management and trap dispatcher routines of the Macintosh operating system. It is provided for those readers who do not have a copy of the *Inside Macintosh* documentation; however, it is not a substitute for obtaining that documentation (or some other description of Macintosh's internal routines). Although it is strongly recommended that the reader be familiar with *Inside Macintosh* or another Macintosh manual if he wants to do any serious debugging, it should also be noted that anyone familiar enough with the Macintosh internals to write a program should be able to debug it!

Memory Map

This is a memory map of the 128K and 512K Macintoshes. All numbers are decimal unless preceded by a dollar sign, in which case they are hexadecimal. @ signs indicate indirection. When there are two addresses given, the bold one stands for the 512K machine, while the light one stands for the 128K machine. RA5 indicates the contents of register A5, which may also be found in location $904.

$FFFFFF		Hardware I/O locations and unused
$410000		ROM
$400000		Unused
$080000/$020000	@$108	Unused RAM
$07FFE4/$01FFE4		Main sound and disk speed buffer
$07FD00/$01FD00		Used by the System Error handler
$07FC80/$01FC80		Main screen buffer
$07A700/$01A700	@$824	

The four addresses below are valid only if the alternate sound and/or screen buffers are used. If these buffers are not used, these memory locations contain the stack and application heap.

$07A3E4/$01A3E4	Alternate sound and disk speed buffer, if used.
$07A100/$01A100	
$077C80/$017C80	Alternate screen buffer, if used.
$072700/$012700	

@$10C		Application program intersegment jump table
RA5+32		Application program parameters
RA5		Application global variables
@RA5		Top of 68000 stack
@$908		Bottom of 68000 stack
RA7		Free memory
@$114		Application heap
@$2AA		System heap
$000B00	@$2A6	System global variables
$000800		A000 trap dispatch table
$000400		System global variables
$000100		68000 exception vectors
$000000		

A000 Trap Dispatcher

The A000 Trap dispatcher is used for almost all ROM calls. ROM routines are called via 68000's illegal opcodes that begin with the bit pattern 1010. The 68000 has a special exception vector for these illegal instructions, which points to a small ROM routine that analyzes the opcode and then uses the dispatch table at $400 to jump to the appropriate routine. The format of the illegal instruction is explained below.

10100abcnnnnnnnnn	Operating system trap. The ROM routine expects its parameters in the 68000 registers (usually; there are exceptions) and preserves all registers except D0, which is usually set to the result or error code. a and b are flags used by some ROM routines. If c is set, the routine affects the A0 register as well as D0. nnnnnnnnn is the routine number.
10101abnnnnnnnnn	Toolbox trap. The ROM routine pulls its parameters on the 68000 stack (usually; there are exceptions) and preserves all registers except D0, D1, D2, A0, and A1. If the routine is a function, the calling program must have decremented the stack pointer by either two or four bytes, depending on the function result size, *before* pushing the parameters. The ROM routine then uses that space to store the function result, and the calling program can then pull the result from the stack by using MOVE (A7)+, *destination*. a must be zero (it was once used as a flag), while b is not used. nnnnnnnnn is a 9-bit routine number.

Memory Management Overview

There are two ways to allocate memory on the Macintosh: on the 68000 stack and on the heaps. The stack is the 68000 supervisor stack; the 68000 user stack is not used. The heaps are managed by a set of ROM routines called the Memory Manager. The heaps are composed of blocks of memory which can be allocated, deallocated, and controlled in other ways by calling ROM routines through A000 traps. Basically, three kinds of blocks may be present on the heap: free, nonrelocatable, and relocatable. Free blocks contain unused memory and are managed entirely by the Memory Manager. Nonrelocatable blocks are addressed by *pointers* (longword 68000 memory addresses). Once allocated, they do not move in memory until they are deallocated. Relocatable blocks, on the other hand, may be frequently moved at the Memory Manager's discretion. Having relocatable blocks is advantageous because allowing the Memory Manager to move blocks allows it to consolidate free blocks when a large blocks has to be allocated. That process is called heap compaction. Since relocatable blocks may be moved on almost any call to the Memory Manager, they can't be addressed by pointers because the pointers would become invalid when the block moved. Instead, they are addressed by *handles*, which are pointers to pointers. The Memory Manager stores addresses of relocatable blocks in a special area of memory (actually in a nonrelocatable blocks) and updates them whenever the blocks move. If the user program keeps pointers to these addresses, or, in other words, handles, they will always remain current. Many bugs are caused by user programs *dereferencing* handles (converting them into pointers) and using those pointers to access the blocks. This practice causes problems if the heap is compacted in the meantime, invalidating the pointers. This is the reasoning that led to the creation of the heap scramble routines in the user area.

Relocatable blocks contain three attributes (bits or flags) which are stored in the upper three bits of the *pointers* to them. Bit 31, if set, "locks" the blocks, making it appear nonrelocatable until that bit is cleared. Bit 30 makes the block "purgeable," which means that if the Memory Manager needs more memory space it can dispose that block. Blocks such as fonts and code segments that are not currently executing are often marked purgeable. Bit 29 is used internally by the Memory and Resource Managers to mark blocks which are resources.

There are two heaps in memory, although the Memory Manager allows the creation of more. The *system heap* has a fixed size (about 16K on 128K Macintoshes, or 46K on 512K Macintoshes), and contains the blocks which hold the important system data structures such as ROM patches, the Chicago 12 font, disk directories, I/O drivers, etc. The *application* heap can be dynamically enlarged and contains the program code, data, resources, other fonts, code segments for desk accessories, and menu, control, or window definition code that is read from the system file.

Use a Heap window to look at the heaps.

Quick Reference

This chapter is a compilation of the frequently used tables in the manual. See the appropriate sections of the manual for more detailed information.

Keys that May be Used in the Monitor

Tab	Move cursor to the top left position of the current window
Return	Process the contents of the line left of the cursor.
Enter	Process the entire line.
Clear (keypad)	Clear the line.
+ (keypad)	Move cursor left.
* (keypad)	Move cursor right.
⌘	⌘A, ⌘B, ⌘D, ⌘E, ⌘F, ⌘G, ⌘H, ⌘M, ⌘N, ⌘P, ⌘R, ⌘S, ⌘T, ⌘O and ⌘U execute commands.
⌘-Shift	⌘-Shift-A, ⌘-Shift-D,⌘-Shift-F, ⌘-Shift-H, and ⌘-Shift-N generate additional windows of the indicated type.

Keys that May be Used outside the Monitor

interrupt	Enters the Monitor.
Option-interrupt	Activates the user area signal function.
⌘-interrupt	Reinitializes the Monitor in emergencies.
Option-⌘-interrupt	A more drastic version of ⌘-interrupt that clears the user area.

Operators Allowed in Expressions

Binary Arithmetic:
+ Addition
- Subtraction
* Multiplication
/ Signed division
\ Signed modulo

Binary Logical:
| Logical OR
^ Logical exclusive OR
& Logical AND

Unary:
+ Positive number
- Negative number
~ Logical NOT
@ Long word indirection
! A000 trap address

Precedence:
< >
unary + - ~ @ !
* / \
&
+ -
^
|

< and > may be used as parentheses.

Register References

variable	value	register name
A0 to A7	RA0 to RA7	Address registers.
D0 to D7	RD0 to RD7	Data registers.
SP*	SP	Same as A7 or RA7. *For anchoring windows only.
SSP*	SSP	System stack pointer. *For anchoring windows only.
USP	USP	User stack pointer. (Normally unused in the Macintosh)
PC	PC	Program counter.
SR		Status register.
CCR		Condition code register.
N	N	The result of the last Number calculation.
V	V	Result of Find, Heap, and other routines.
	USER	The beginning of the user area.
	DSPT	The address of the ROM A000 trap dispatcher.

Quick Reference

This chapter is a compilation of the frequently used tables in the manual. See the appropriate sections of the manual for more detailed information.

Keys that May be Used in the Monitor

Tab	Move cursor to the top left position of the current window
Return	Process the contents of the line left of the cursor.
Enter	Process the entire line.
Clear (keypad)	Clear the line.
+ (keypad)	Move cursor left.
* (keypad)	Move cursor right.
⌘	⌘A, ⌘B, ⌘D, ⌘E, ⌘F, ⌘G, ⌘H, ⌘M, ⌘N, ⌘P, ⌘R, ⌘S, ⌘T, ⌘O and ⌘U execute commands.
⌘-Shift	⌘-Shift-A, ⌘-Shift-D, ⌘-Shift-F, ⌘-Shift-H, and ⌘-Shift-N generate additional windows of the indicated type.

Keys that May be Used outside the Monitor

Interrupt	Enters the Monitor.
Option-Interrupt	Activates the user area signal function.
⌘-Interrupt	Reinitializes the Monitor in emergencies.
Option-⌘-interrupt	A more drastic version of ⌘-interrupt that clears the user area.

Operators Allowed in Expressions

Binary Arithmetic:	Binary Logical:	Unary:	Precedence:
+ Addition		+ Positive number	< >
− Subtraction		Logical OR	− Negative number	unary + − ~ @ !
* Multiplication	& Logical AND	~ Logical NOT	* / \
/ Signed division		@ Long word indirection	;
\ Signed modulo		! A000 trap address	+ −
			v

< and > may be used as parentheses.

Register References

variable	value	register name
A0 to A7	RA0 to RA7	Address registers.
D0 to D7	RD0 to RD7	Data registers.
SP*	SP	Same as A7 or RA7. *For anchoring windows only.
SSP*	SSP	System stack pointer. *For anchoring windows only.
USP	USP	User stack pointer. (Normally unused in the Macintosh)
PC	PC	Program counter.
SR		Status register.
CCR		Condition code register.
N	N	The result of the last Number calculation.
V	V	Result of Find, Heap, and other routines.
USER		The beginning of the user area.
DSPT		The address of the ROM A000 trap dispatcher.

Dump Window Flags

P	Program counter		*	Breakpoint
S	System stack pointer		N	N register
U	User stack pointer		V	V register
0 to 6	Address register			

Assembly Window Addressing Modes

Dn	Data register direct
An	Address register direct
(An)	Register indirect
(An)+	Postincrement register indirect
-(An)	Predecrement register indirect
offset(An)	Register indirect with offset
offset(An,Rns)	Register indirect with offset and index
address	Absolute
^*address*, *, *+*offset*, or *-*offset*	Relative with offset
^*address*(Rns), *(Rns), *+*offset*(Rns), or *+*offset*(Rns)	Relative with offset and index
#number	Immediate
USP, SR, CCR	Implied register

Rns is an abbreviation for either a data or an address register optionally followed by either a word or longword size indication. Some examples of Rns are D0, A0, D3.W, and A7.L.

Items Identified by the Heap Window

GrayRgn	$9EE	The rounded region defining the desktop.
MenuList	$A1C	The current menu bar list.
TEScrap	$AB4	TextEdit scrap.
Scrap	$964	Memory scrap.
SaveVisRgn	$9F2	A region used by the Window Manager.
WmgrPort	$9DE	A grafPort used by the Window Manager.
FinderInfo	CurrentA5+$10	The Finder information handle (in system heap).
ParamText0	$AA0	The first parameter in the last ParamText call.
ParamText1	$AA4	The second parameter in the last ParamText call.
ParamText2	$AA8	The third parameter in the last ParamText call.
ParamText3	$AAC	The fourth parameter in the last ParamText call.
Resource map $..		Resource map of the given resource file.
Window #$.., kind $..		A window found by following the window list. The first number is the number of the window (0 is the frontmost window, 1 the next one, etc). The second number is the value of windowKind for that window.

(The hexadecimal numbers are either handles or pointers to the items above)

In addition to the items listed above, the components of WmgrPort and all windows on the window list are identified. These components are identified first by either by the phrase (Window @$..), which indicates the location of the window to which they belong, or (WmgrPort), indicating that they belong to the WmgrPort. After one of these two identifications one of the following messages is displayed:

VisRgn	The region of the window which is visible on the screen.
ClipRgn	The clipping region.
PicSave	Data for a picture being saved.

RgnSave	Data for a region being saved.
PolySave	Data for a polygon being saved.
StrucRgn	Structure region of window.
ContRgn	Content region of window.
UpdateRgn	Update region of window.
WData	Window-defined data.
WTitle	The title string.
WPic	Window's picture used for updating.
DlgItemList	Item list (dialog windows only).
TEHandle	TextEdit record (dialog windows only).
Item #$.. type $..	An item in the window's DlgItemList (dialog windows only). The first number is the item number (first item is 0, second 1, etc.), and the second number is the item type.
Control	Any control belonging to the window. This is displayed only if the control could not be identified as an item in that window's dialog list.

Heap Window Handle Flags

l L   Locked   p P   Purgeable   r R   Resource

File Window Map Flags

r R   Read-only map   c C   Map will be compacted   w W   Map will be written to disk

File Window Resource Flags

. R   System reference (opposite is local reference)
. H   Load into system heap (opposite is application heap)
. P   Purgeable
. L   Locked
. T   Protected
. l   Preloaded
. W   Write into resource file
. U   U flag set

A000 Trap Names and Numbers in Numerical Order

$A000:	_Open	$A001:	_Close	$A002:	_Read	$A003:	_Write
$A004:	_Control	$A005:	_Status	$A006:	_KillIO	$A007:	_GetVolInfo
$A008:	_Create	$A009:	_Delete	$A00A:	_OpenRF	$A00B:	_Rename
$A00C:	_GetFileInfo	$A00D:	_SetFileInfo	$A00E:	_UnmountVol	$A00F:	_MountVol
$A010:	_Allocate	$A011:	_GetEOF	$A012:	_SetEOF	$A013:	_FlushVol
$A014:	_GetVol	$A015:	_SetVol	$A016:	_InitQueue	$A017:	_Eject
$A018:	_GetFPos	$A019:	_InitZone	$A11A:	_GetZone	$A01B:	_SetZone
$A01C:	_FreeMem	$A11D:	_MaxMem	$A11E:	_NewPtr	$A01F:	_DisposPtr
$A020:	_SetPtrSize	$A021:	_GetPtrSize	$A122:	_NewHandle	$A023:	_DisposHandle
$A024:	_SetHandleSize	$A025:	_GetHandleSize	$A126:	_HandleZone	$A027:	_ReallocHandl
$A128:	_RecoverHandle	$A029:	_HLock	$A02A:	_HUnlock	$A02B:	_EmptyHandle
$A02C:	_InitApplZone	$A02D:	_SetApplLimit	$A02E:	_BlockMove	$A02F:	_PostEvent
$A030:	_OSEventAvail	$A031:	_GetOSEvent	$A032:	_FlushEvents	$A033:	_VInstall
$A034:	_VRemove	$A035:	_OffLine	$A036:	_MoreMasters	$A037:	_ReadParam
$A03B:	_WriteParam	$A039:	_ReadDateTime	$A03A:	_SetDateTime	$A03B:	_Delay
$A03C:	_CmpString	$A03D:	_DrvrInstall	$A03E:	_DrvrRemove	$A03F:	_InitUtil
$A140:	_ResrvMem	$A041:	_SetFilLock	$A042:	_RstFilLock	$A043:	_SetFilType
$A044:	_SetFPos	$A045:	_FlushFile	$A146:	_GetTrapAddress	$A047:	_SetTra 'dre

$A148:	_PtrZone	$A049:	_HPurge	$A04A:	_HNoPurge	$A04B:	_SetGrowZone
$A14C:	_CompactMem	$A14D:	_PurgeMem	$A04E:	_AddDrive	$A04F:	_RDrvrInstall
$A850:	_InitCursor	$A851:	_SetCursor	$A852:	_HideCursor	$A853:	_ShowCu:
$A054:	_UprString	$A855:	_ShieldCursor	$A856:	_ObscureCursor	$A057:	_SetAppBase
$A858:	_BitAnd	$A859:	_BitXor	$A85A:	_BitNot	$A85B:	_BitOr
$A85C:	_BitShift	$A85D:	_BitTst	$A85E:	_BitSet	$A85F:	_BitClr
$A860:		$A861:	_Random	$A862:	_ForeColor	$A863:	_BackColor
$A864:	_ColorBit	$A865:	_GetPixel	$A866:	_StuffHex	$A867:	_LongMul
$A868:	_FixMul	$A869:	_FixRatio	$A86A:	_HiWord	$A86B:	_LoWord
$A86C:	_FixRound	$A86D:	_InitPort	$A86E:	_InitGraf	$A86F:	_OpenPort
$A870:	_LocalToGlobal	$A871:	_GlobalToLocal	$A872:	_GrafDevice	$A873:	_SetPort
$A874:	_GetPort	$A875:	_SetPBits	$A876:	_PortSize	$A877:	_MovePortTo
$A878:	_SetOrigin	$A879:	_SetClip	$A87A:	_GetClip	$A87B:	_ClipRect
$A87C:	_BackPat	$A87D:	_ClosePort	$A87E:	_AddPt	$A87F:	_SubPt
$A880:	_SetPt	$A881:	_EqualPt	$A882:	_StdText	$A883:	_DrawChar
$A884:	_DrawString	$A885:	_DrawText	$A886:	_TextWidth	$A887:	_TextFont
$A888:	_TextFace	$A889:	_TextMode	$A88A:	_TextSize	$A88B:	_GetFontInfo
$A88C:	_StringWidth	$A88D:	_CharWidth	$A88E:	_SpaceExtra	$A88F:	
$A890:	_StdLine	$A891:	_LineTo	$A892:	_Line	$A893:	_MoveTo
$A894:	_Move	$A895:		$A896:	_HidePen	$A897:	_ShowPen
$A898:	_GetPenState	$A899:	_SetPenState	$A89A:	_GetPen	$A89B:	_PenSize
$A89C:	_PenMode	$A89D:	_PenPat	$A89E:	_PenNormal	$A89F:	
$A8A0:	_StdRect	$A8A1:	_FrameRect	$A8A2:	_PaintRect	$A8A3:	_EraseRect
$A8A4:	_InverRect	$A8A5:	_FillRect	$A8A6:	_EqualRect	$A8A7:	_SetRect
$A8A8:	_OffsetRect	$A8A9:	_InsetRect	$A8AA:	_SectRect	$A8AB:	_UnionRect
$A8AC:	_Pt2Rect	$A8AD:	_PtInRect	$A8AE:	_EmptyRect	$A8AF:	_StdRRect
$A8B0:	_FrameRoundRect	$A8B1:	_PaintRoundRect	$A8B2:	_EraseRoundRect	$A8B3:	_InverRoundRe
$A8B4:	_FillRoundRect	$A8B5:		$A8B6:	_StdOval	$A8B7:	_FrameOval
$A8B8:	_PaintOval	$A8B9:	_EraseOval	$A8BA:	_InvertOval	$A8BB:	_FillOval
$A8BC:	_SlopeFromAngle	$A8BD:	_StdArc	$A8BE:	_FrameArc	$A8BF:	_PaintArc
$A8C0:	_EraseArc	$A8C1:	_InvertArc	$A8C2:	_FillArc	$A8C3:	_PtToAngle
$A8C4:	_AngleFromSlope	$A8C5:	_StdPoly	$A8C6:	_FramePoly	$A8C7:	_PaintPoly
$A8C8:	_ErasePoly	$A8C9:	_InvertPoly	$A8CA:	_FillPoly	$A8CB:	_OpenPoly
$A8CC:	_ClosePoly	$A8CD:	_KillPoly	$A8CE:	_OffsetPoly	$A8CF:	_PackBits
$A8D0:	_UnpackBits	$A8D1:	_StdRgn	$A8D2:	_FrameRgn	$A8D3:	_PaintRgn
$A8D4:	_EraseRgn	$A8D5:	_InverRgn	$A8D6:	_FillRgn	$A8D7:	
$A8D8:	_NewRgn	$A8D9:	_DisposRgn	$A8DA:	_OpenRgn	$A8DB:	_CloseRgn
$A8DC:	_CopyRgn	$A8DD:	_SetEmptyRgn	$A8DE:	_SetRecRgn	$A8DF:	_RectRgn
$A8E0:	_OffsetRgn	$A8E1:	_InsetRgn	$A8E2:	_EmptyRgn	$A8E3:	_EqualRgn
$A8E4:	_SectRgn	$A8E5:	_UnionRgn	$A8E6:	_DiffRgn	$A8E7:	_XorRgn
$A8E8:	_PtInRgn	$A8E9:	_RectInRgn	$A8EA:	_SetStdProcs	$A8EB:	_StdBits
$A8EC:	_CopyBits	$A8ED:	_StdTxMeasure	$A8EE:	_StdGetPic	$A8EF:	_ScrollRect
$A8F0:	_StdPutPic	$A8F1:	_StdComment	$A8F2:	_PicComment	$A8F3:	_OpenPicture
$A8F4:	_ClosePicture	$A8F5:	_KillPicture	$A8F6:	_DrawPicture	$A8F7:	
$A8F8:	_ScalePt	$A8F9:	_MapPt	$A8FA:	_MapRect	$A8FB:	_MapRgn
$A8FC:	_MapPoly	$A8FD:		$A8FE:	_InitFonts	$A8FF:	_GetFName
$A900:	_GetFNum	$A901:	_FMSwapFont	$A902:	_RealFont	$A903:	_SetFontLock
$A904:	_DrawGrowIcon	$A905:	_DragGrayRgn	$A906:	_NewString	$A907:	_SetString
$A908:	_ShowHide	$A909:	_CalcVis	$A90A:	_CalcVBehind	$A90B:	_ClipAbove
$A90C:	_PaintOne	$A90D:	_PaintBehind	$A90E:	_SaveOld	$A90F:	_DrawNew
$A910:	_GetWMgrPort	$A911:	_CheckUpdate	$A912:	_InitWindows	$A913:	_NewWindow
$A914:	_DisposWindow	$A915:	_ShowWindow	$A916:	_HideWindow	$A917:	_GetWRefCon
$A918:	_SetWRefCon	$A919:	_GetWTitle	$A91A:	_SetWTitle	$A91B:	_MoveWindow
$A91C:	_HiliteWindow	$A91D:	_SizeWindow	$A91E:	_TrackGoAway	$A91F:	_SelectWindow
$A920:	_BringToFront	$A921:	_SendBehind	$A922:	_BeginUpdate	$A923:	_EndUpdate
$A924:	_FrontWindow	$A925:	_DragWindow	$A926:	_DragTheRgn	$A927:	_InvalRgn
$A928:	_InvalRect	$A929:	_ValidRgn	$A92A:	_ValidRect	$A92B:	_GrowWind
$A92C:	_FindWindow	$A92D:	_CloseWindow	$A92E:	_SetWindowPic	$A92F:	_GetWindo...c
$A930:	_InitMenus	$A931:	_NewMenu	$A932:	_DisposMenu	$A933:	_AppendMenu
$A934:	_ClearMenuBar	$A935:	_InsertMenu	$A936:	_DeleteMenu	$A937:	_DrawMenuBar

$A938: _HiliteMenu	$A939: _EnableItem	$A93A: _DisableItem	$A93B: _GetMenuBar
$A93C: _SetMenuBar	$A93D: _MenuSelect	$A93E: _MenuKey	$A93F: _GetItmIcon
$A940: _SetItmIcon	$A941: _GetItmStyle	$A942: _SetItmStyle	$A943: _GetItmMark
$A944: _SetItmMark	$A945: _CheckItem	$A946: _GetItem	$A947: _SetItem
$A948: _CalcMenuSize	$A949: _GetMHandle	$A94A: _SetMFlash	$A94B: _PlotIco
$A94C: _FlashMenuBar	$A94D: _AddResMenu	$A94E: _PinRect	$A94F: _DeltaPoint
$A950: _CountMItems	$A951: _InsertResMenu	$A952:	$A953:
$A954: _NewControl	$A955: _DisposControl	_KillControls	$A957: _ShowControl
$A958: _HideControl	$A959: _MoveControl	$A956: _GetCRefCon	$A95B: _SetCRefCon
$A95C: _SizeControl	$A95D: _HiliteControl	$A95A: _GetCTitle	$A95F: _SetCTitle
$A960: _GetCtlValue	$A961: _GetMinCtl	$A95E: _GetMaxCtl	$A963: _SetCtlValue
$A964: _SetMinCtl	$A965: _SetMaxCtl	$A962: _TestControl	$A967: _DragControl
$A968: _TrackControl	$A969: _DrawControls	$A966: _GetCtlAction	$A96B: _SetCtlAction
$A96C: _FindControl	$A96D:	$A96A: _Dequeue	$A96F: _Enqueue
$A970: _GetNextEvent	$A971: _EventAvail	$A96E: _GetMouse	$A973: _StillDown
$A974: _Button	$A975: _TickCount	$A972: _GetKeys	$A977: _WaitMouseUp
$A978:	$A979: _CouldDialog	$A976: _FreeDialog	$A97B: _InitDialogs
$A97C: _GetNewDialog	$A97D: _NewDialog	$A97A: _SelIText	$A97F: _IsDialogEven
$A980: _DialogSelect	$A981: _DrawDialog	$A97E: _CloseDialog	$A983: _DisposDialog
$A984:	$A985: _Alert	$A982: _StopAlert	$A987: _NoteAlert
$A988: _CautionAlert	$A989: _CouldAlert	$A986: _FreeAlert	$A98B: _ParamText
$A98C: _ErrorSound	$A98D: _GetDItem	$A98A: _SetDItem	$A98F: _SetIText
$A990: _GetIText	$A991: _ModalDialog	$A98E: _DetachResource	$A993: _SetResPurge
$A994: _CurResFile	$A995: _InitResources	$A992: _RsrcZoneInit	$A997: _OpenResFile
$A998: _UseResFile	$A999: _UpdateResFile	$A996: _CloseResFile	$A99B: _SetResLoad
$A99C: _CountResources	$A99D: _GetIndResource	$A99A: _CountTypes	$A99F: _GetIndType
$A9A0: _GetResource	$A9A1: _GetNamedResource	$A99E: _LoadResource	$A9A3: _ReleaseResou
$A9A4: _HomeResFile	$A9A5: _SizeResource	$A9A2: _GetResAttrs	$A9A7: _SetResAttrs
$A9A8: _GetResInfo	$A9A9: _SetResInfo	$A9A6: _ChangedResource	$A9AB: _AddResource
$A9AC: _AddReference	$A9AD: _RmveResource	$A9AA: _RmveRefence	$A9AF: _ResError
$A9B0: _WriteResource	$A9B1: _CreateResFile	$A9AE: _SystemEvent	$A9B3: _SystemClick
$A9B4: _SystemTask	$A9B5: _SystemMenu	$A9B2: _OpenDeskAcc	$A9B7: _CloseD Acc
$A9B8: _GetPattern	$A9B9: _GetCursor	$A9B6: _GetString	$A9BB: _GetIcon
$A9BC: _GetPicture	$A9BD: _GetNewWindow	$A9BA: _GetNewControl	$A9BF: _GetRMenu
$A9C0: _GetNewMBar	$A9C1: _UniqueID	$A9BE: _SysEdit	$A9C3:
$A9C4:	$A9C5:	$A9C2: _Secs2Date	$A9C7: _Date2Secs
$A9C8: _SysBeep	$A9C9: _SysError	$A9C6: _PutIcon	$A9CB: _TEGetText
$A9CC: _TEInit	$A9CD: _TEDispose	$A9CA: _TextBox	$A9CF: _TESetText
$A9D0: _TECalText	$A9D1: _TESetSelect	$A9CE: _TENew	$A9D3: _TEUpdate
$A9D4: _TEClick	$A9D5: _TECopy	$A9D2: _TECut	$A9D7: _TEDelete
$A9D8: _TEActivate	$A9D9: _TEDeactivate	$A9D6: _TEIdle	$A9DB: _TEPaste
$A9DC: _TEKey	$A9DD: _TEScroll	$A9DA: _TEInsert	$A9DF: _TESetJust
$A9E0: _Munger	$A9E1: _HandToHand	$A9DE: _PtrToXHand	$A9E3: _PtrToHand
$A9E4: _HandAndHand	$A9E5: _InitPack	$A9E2: _InitMath	$A9E7: _Pack0
$A9E8: _Pack1	$A9E9: _Pack2	$A9E6: _Pack3	$A9EB: _FP68K
$A9EC: _Elems68K	$A9ED: _Pack6	$A9EA: _Pack7	$A9EF: _PtrAndHand
$A9F0: _LoadSeg	$A9F1: _UnloadSeg	$A9EE: _Launch	$A9F3: _Chain
$A9F4: _ExitToShell	$A9F5: _GetAppParms	$A9F2: _GetResFileAttrs	$A9F7: _SetResFileAt
$A9F8:	$A9F9: _InfoScrap	$A9F6: _UnlodeScrap	$A9FB: _LodeScrap
$A9FC: _ZeroScrap	$A9FD: _GetScrap	$A9FA: _PutScrap	$A9FF: _Debugger
		$A9FE:	

A000 Trap Names and Numbers in Alphabetical Order

$A04E: _AddDrive	$A87E: _AddPt	$A9AC: _AddReference	$A94D: _AddResMenu
$A9AB: _AddResource	$A985: _Alert	$A010: _Allocate	$A8C4: _AngleFromSlo
$A933: _AppendMenu	$A863: _BackColor	$A87C: _BackPat	$A922: _BeginUpdate
$A858: _BitAnd	$A85F: _BitClr	$A85A: _BitNot	$A85B: _BitOr
$A85E: _BitSet	$A85C: _BitShift	$A85D: _BitTst	$A859: _BitXor
$A02E: _BlockMove	$A920: _BringToFront	$A974: _Button	$A948: _CalcM Size

$A90A: _CalcVBehind	$A909: _CalcVis	$A988: _CautionAlert	$A9F3: _Chain
$A9AA: _ChangedResource	$A88D: _CharWidth	$A945: _CheckItem	$A911: _CheckUpdate
$A934: _ClearMenuBar	$A90B: _ClipAbove	$A87B: _ClipRect	$A001: _Close
$A9B7: _CloseDeskAcc	$A982: _CloseDialog	$A8F4: _ClosePicture	$A8CC: _ClosePo_
$A87D: _ClosePort	$A99A: _CloseResFile	$A8DB: _CloseRgn	$A92D: _CloseWindow
$A03C: _CmpString	$A864: _ColorBit	$A14C: _CompactMem	$A004: _Control
$A8EC: _CopyBits	$A8DC: _CopyRgn	$A989: _CouldAlert	$A979: _CouldDialog
$A950: _CountMItems	$A99C: _CountResources	$A99E: _CountTypes	$A008: _Create
$A9B1: _CreateResFile	$A994: _CurResFile	$A9C7: _Date2Secs	$A9FF: _Debugger
$A03B: _Delay	$A009: _Delete	$A936: _DeleteMenu	$A94F: _DeltaPoint
$A96E: _Dequeue	$A992: _DetachResource	$A980: _DialogSelect	$A8E6: _DiffRgn
$A93A: _DisableItem	$A955: _DisposControl	$A983: _DisposDialog	$A023: _DisposHandle
$A932: _DisposMenu	$A01F: _DisposPtr	$A8D9: _DisposRgn	$A914: _DisposWindow
$A967: _DragControl	$A905: _DragGrayRgn	$A926: _DragTheRgn	$A925: _DragWindow
$A883: _DrawChar	$A969: _DrawControls	$A981: _DrawDialog	$A904: _DrawGrowIcon
$A937: _DrawMenuBar	$A90F: _DrawNew	$A8F6: _DrawPicture	$A884: _DrawString
$A885: _DrawText	$A03D: _DrvrInstall	$A03E: _DrvrRemove	$A017: _Eject
$A9EC: _Elems68K	$A02B: _EmptyHandle	$A8AE: _EmptyRect	$A8E2: _EmptyRgn
$A939: _EnableItem	$A923: _EndUpdate	$A96F: _Enqueue	$A881: _EqualPt
$A8A6: _EqualRect	$A8E3: _EqualRgn	$A8C0: _EraseArc	$A8B9: _EraseOval
$A8C8: _ErasePoly	$A8A3: _EraseRect	$A8D4: _EraseRgn	$A8B2: _EraseRoundRe
$A98C: _ErrorSound	$A971: _EventAvail	$A9F4: _ExitToShell	$A901: _FMSwapFont
$A9EB: _FP68K	$A8C2: _FillArc	$A8BB: _FillOval	$A8CA: _FillPoly
$A8A5: _FillRect	$A8D6: _FillRgn	$A8B4: _FillRoundRect	$A96C: _FindControl
$A92C: _FindWindow	$A868: _FixMul	$A869: _FixRatio	$A86C: _FixRound
$A94C: _FlashMenuBar	$A032: _FlushEvents	$A045: _FlushFile	$A013: _FlushVol
$A862: _ForeColor	$A8BE: _FrameArc	$A8B7: _FrameOval	$A8C6: _FramePoly
$A8A1: _FrameRect	$A8D2: _FrameRgn	$A8B0: _FrameRoundRect	$A98A: _FreeAlert
$A97A: _FreeDialog	$A01C: _FreeMem	$A924: _FrontWindow	$A9F5: _GetAppParms
$A95A: _GetCRefCon	$A95E: _GetCTitle	$A87A: _GetClip	$A96A: _GetCtlAction
$A960: _GetCtlValue	$A9B9: _GetCursor	$A98D: _GetDItem	$A011: _GetEOF
$A8FF: _GetFName	$A900: _GetFNum	$A018: _GetFPos	$A00C: _GetFileInfo
$A88B: _GetFontInfo	$A025: _GetHandleSize	$A990: _GetIText	$A9BB: _GetIcon
$A99D: _GetIndResource	$A99F: _GetIndType	$A946: _GetItem	$A93F: _GetItmIcon
$A943: _GetItmMark	$A941: _GetItmStyle	$A976: _GetKeys	$A949: _GetMHandle
$A962: _GetMaxCtl	$A93B: _GetMenuBar	$A961: _GetMinCtl	$A972: _GetMouse
$A9A1: _GetNamedResource	$A9BE: _GetNewControl	$A97C: _GetNewDialog	$A9C0: _GetNewMBar
$A9BD: _GetNewWindow	$A970: _GetNextEvent	$A031: _GetOSEvent	$A9B8: _GetPattern
$A89A: _GetPen	$A898: _GetPenState	$A9BC: _GetPicture	$A865: _GetPixel
$A874: _GetPort	$A021: _GetPtrSize	$A9BF: _GetRMenu	$A9A6: _GetResAttrs
$A9F6: _GetResFileAttrs	$A9A8: _GetResInfo	$A9A0: _GetResource	$A9FD: _GetScrap
$A9BA: _GetString	$A146: _GetTrapAddress	$A014: _GetVol	$A007: _GetVolInfo
$A910: _GetWMgrPort	$A917: _GetWRefCon	$A919: _GetWTitle	$A92F: _GetWindowPic
$A11A: _GetZone	$A871: _GlobalToLocal	$A872: _GrafDevice	$A92B: _GrowWindow
$A029: _HLock	$A04A: _HNoPurge	$A049: _HPurge	$A02A: _HUnlock
$A9E4: _HandAndHand	$A9E1: _HandToHand	$A126: _HandleZone	$A86A: _HiWord
$A958: _HideControl	$A852: _HideCursor	$A896: _HidePen	$A916: _HideWindow
$A95D: _HiliteControl	$A938: _HiliteMenu	$A91C: _HiliteWindow	$A9A4: _HomeResFile
$A9F9: _InfoScrap	$A02C: _InitApplZone	$A850: _InitCursor	$A97B: _InitDialogs
$A8FE: _InitFonts	$A86E: _InitGraf	$A9E6: _InitMath	$A930: _InitMenus
$A9E5: _InitPack	$A86D: _InitPort	$A016: _InitQueue	$A995: _InitResource
$A03F: _InitUtil	$A912: _InitWindows	$A019: _InitZone	$A935: _InsertMenu
$A951: _InsertResMenu	$A8A9: _InsetRect	$A8E1: _InsetRgn	$A928: _InvalRect
$A927: _InvalRgn	$A8A4: _InverRect	$A8D5: _InverRgn	$A8B3: _InverRoundRe
$A8C1: _InvertArc	$A8BA: _InvertOval	$A8C9: _InvertPoly	$A97F: _IsDialogEven
$A956: _KillControls	$A006: _KillIO	$A8F5: _KillPicture	$A8CD: _KillPoly
$A9F2: _Launch	$A892: _Line	$A891: _LineTo	$A86B: _LoWord
$A9A2: _LoadResource	$A9F0: _LoadSeg	$A870: _LocalToGlobal	$A9FB: _LodeScra
$A867: _LongMul	$A8FC: _MapPoly	$A8F9: _MapPt	$A8FA: _MapRect
$A8FB: _MapRgn	$A11D: _MaxMem	$A93E: _MenuKey	$A93D: _MenuSelect

$A991: _ModalDialog	$A036: _MoreMasters	$A00F: _MountVol	$A894: _Move
$A959: _MoveControl	$A877: _MovePortTo	$A893: _MoveTo	$A91B: _MoveWindow
$A9E0: _Munger	$A954: _NewControl	$A97D: _NewDialog	$A122: _NewHandle
$A931: _NewMenu	$A11E: _NewPtr	$A8D8: _NewRgn	$A906: _NewStrin~
$A913: _NewWindow	$A987: _NoteAlert	$A030: _OSEventAvail	$A856: _Obscure(;o
$A035: _OffLine	$A8CE: _OffsetPoly	$A8A8: _OffsetRect	$A8E0: _OffsetRgn
$A000: _Open	$A9B6: _OpenDeskAcc	$A8F3: _OpenPicture	$A8CB: _OpenPoly
$A86F: _OpenPort	$A00A: _OpenRF	$A997: _OpenResFile	$A8DA: _OpenRgn
$A9E7: _Pack0	$A9E8: _Pack1	$A9E9: _Pack2	$A9EA: _Pack3
$A9ED: _Pack6	$A9EE: _Pack7	$A8CF: _PackBits	$A8BF: _PaintArc
$A90D: _PaintBehind	$A90C: _PaintOne	$A8B8: _PaintOval	$A8C7: _PaintPoly
$A8A2: _PaintRect	$A8D3: _PaintRgn	$A8B1: _PaintRoundRect	$A98B: _ParamText
$A89C: _PenMode	$A89E: _PenNormal	$A89D: _PenPat	$A89B: _PenSize
$A8F2: _PicComment	$A94E: _PinRect	$A94B: _PlotIcon	$A876: _PortSize
$A02F: _PostEvent	$A8AC: _Pt2Rect	$A8AD: _PtInRect	$A8E8: _PtInRgn
$A8C3: _PtToAngle	$A9EF: _PtrAndHand	$A9E3: _PtrToHand	$A9E2: _PtrToXHand
$A148: _PtrZone	$A14D: _PurgeMem	$A9CA: _PutIcon	$A9FE: _PutScrap
$A04F: _RDrvrInstall	$A861: _Random	$A002: _Read	$A039: _ReadDateTime
$A037: _ReadParam	$A902: _RealFont	$A027: _ReallocHandle	$A128: _RecoverHandl
$A8E9: _RectInRgn	$A8DF: _RectRgn	$A9A3: _ReleaseResource	$A00B: _Rename
$A9AF: _ResError	$A140: _ResrvMem	$A9AE: _RmveRefence	$A9AD: _RmveResource
$A996: _RsrcZoneInit	$A042: _RstFilLock	$A90E: _SaveOld	$A8F8: _ScalePt
$A8EF: _ScrollRect	$A9C6: _Secs2Date	$A8AA: _SectRect	$A8E4: _SectRgn
$A97E: _SelIText	$A91F: _SelectWindow	$A921: _SendBehind	$A057: _SetAppBase
$A02D: _SetApplLimit	$A95B: _SetCRefCon	$A95F: _SetCTitle	$A879: _SetClip
$A96B: _SetCtlAction	$A963: _SetCtlValue	$A851: _SetCursor	$A98E: _SetDItem
$A03A: _SetDateTime	$A012: _SetEOF	$A8DD: _SetEmptyRgn	$A044: _SetFPos
$A041: _SetFilLock	$A043: _SetFilType	$A00D: _SetFileInfo	$A903: _SetFontLock
$A04B: _SetGrowZone	$A024: _SetHandleSize	$A98F: _SetIText	$A947: _SetItem
$A940: _SetItmIcon	$A944: _SetItmMark	$A942: _SetItmStyle	$A94A: _SetMFlash
$A965: _SetMaxCtl	$A93C: _SetMenuBar	$A964: _SetMinCtl	$A878: _SetOrigin
$A875: _SetPBits	$A899: _SetPenState	$A873: _SetPort	$A880: _SetPt
$A020: _SetPtrSize	$A8DE: _SetRecRgn	$A8A7: _SetRect	$A9A7: _SetResAttrs
$A9F7: _SetResFileAttrs	$A9A9: _SetResInfo	$A99B: _SetResLoad	$A993: _SetResPurge
$A8EA: _SetStdProcs	$A907: _SetString	$A047: _SetTrapAddress	$A015: _SetVol
$A918: _SetWRefCon	$A91A: _SetWTitle	$A92E: _SetWindowPic	$A01B: _SetZone
$A855: _ShieldCursor	$A957: _ShowControl	$A853: _ShowCursor	$A908: _ShowHide
$A897: _ShowPen	$A915: _ShowWindow	$A95C: _SizeControl	$A9A5: _SizeResource
$A91D: _SizeWindow	$A8BC: _SlopeFromAngle	$A88E: _SpaceExtra	$A005: _Status
$A8BD: _StdArc	$A8EB: _StdBits	$A8F1: _StdComment	$A8EE: _StdGetPic
$A890: _StdLine	$A8B6: _StdOval	$A8C5: _StdPoly	$A8F0: _StdPutPic
$A8AF: _StdRRect	$A8A0: _StdRect	$A8D1: _StdRgn	$A882: _StdText
$A8ED: _StdTxMeasure	$A973: _StillDown	$A986: _StopAlert	$A88C: _StringWidth
$A866: _StuffHex	$A87F: _SubPt	$A9C8: _SysBeep	$A9C2: _SysEdit
$A9C9: _SysError	$A9B3: _SystemClick	$A9B2: _SystemEvent	$A9B5: _SystemMenu
$A9B4: _SystemTask	$A9D8: _TEActivate	$A9D0: _TECalText	$A9D4: _TEClick
$A9D5: _TECopy	$A9D6: _TECut	$A9D9: _TEDeactivate	$A9D7: _TEDelete
$A9CD: _TEDispose	$A9CB: _TEGetText	$A9DA: _TEIdle	$A9CC: _TEInit
$A9DE: _TEInsert	$A9DC: _TEKey	$A9D2: _TENew	$A9DB: _TEPaste
$A9DD: _TEScroll	$A9DF: _TESetJust	$A9D1: _TESetSelect	$A9CF: _TESetText
$A9D3: _TEUpdate	$A966: _TestControl	$A9CE: _TextBox	$A888: _TextFace
$A887: _TextFont	$A889: _TextMode	$A88A: _TextSize	$A886: _TextWidth
$A975: _TickCount	$A968: _TrackControl	$A91E: _TrackGoAway	$A8AB: _UnionRect
$A8E5: _UnionRgn	$A9C1: _UniqueID	$A9F1: _UnloadSeg	$A9FA: _UnlodeScrap
$A00E: _UnmountVol	$A8D0: _UnpackBits	$A999: _UpdateResFile	$A054: _UprString
$A998: _UseResFile	$A033: _VInstall	$A034: _VRemove	$A92A: _ValidRect
$A929: _ValidRgn	$A977: _WaitMouseUp	$A003: _Write	$A038: _WriteParam
$A9B0: _WriteResource	$A8E7: _XorRgn	$A9FC: _ZeroScrap	

Conclusion

I hope that you have found this manual and the product it describes useful in your work. I am open to any suggestions on improving the Monitor or the manual. I would especially like to see examples of user areas more powerful than the ones that I have provided.

<div align="right">Waldemar Horwat</div>

ICOM Simulations, Inc.
626 Wheeling Road
Wheeling, IL 60090
(312) 520-4443

Note: TMQ Software, Inc. were the originators of TMON. The product is now being distributed & supported by ICOM Simulations, Inc.